# java基础

## 1.1 Java普通类、接口和抽象类的比较；

### 1.1.1 普通类和抽象类的比较：

1. 普通类可以被实例化，抽象类不可以被实例化；
2. 当一个子类继承了抽象类之后，子类必须要实现抽象类的抽象方法，除非该子类也是一个抽象类；
3. 抽象方法不能是static修饰，因为static属于字节码层面，是可以不用实例化对象就可以运行的，而抽象方法又没有方法体，所以无法运行；

### 1.1.2 抽象类和Java接口的比较：

1. 抽象类和接口都不能被实例化；
2. 接口中除了default修饰的方法外，其它方法均不能有方法体（此特性仅仅在jdk1.8后适用）；而抽象类中的方法可以有方法体；
3. 实现接口用implements，继承抽象类用extends，并且继承是单继承，而实现可以是多实现；
4. 接口成员变量默认是public static final ，必须赋初值，且不能被修改，方法默认是public abstract；而抽象类中方法和变量则默认是default，抽象方法需要用abstract修饰；

## 1.2 方法的重写和重载；

### 1.2.1 重写：

1.重写是指子类重写父类的方法；

2.参数列表必须和父类的方法完全一样；

3.返回类型必须是父类方法返回类型的派生类，或者和父类方法返回类型一样；

4.访问权限不能被父类方法的访问权限更低；

5.所抛出的异常必须是父类方法抛出异常的派生类，或者和父类方法抛出的异常相同；

6.如果父类的方法被static修饰，则子类不可以继承，但是子类可以重新声明该方法；

### 1.2.2 重载：

1.重载是指在一个类中方法名字相同，而参数列表不同。返回类型可以相同，也可以不相同；

2.重载方法可以改变访问权限；

3.重载方法可以声明更加广泛的检查异常；

4.子类可以重载父类的方法；

## 1.3 重要的关键字

### 1.3.1 final：

1.final用于修饰各种类型的常量；

2.final修饰的class无法被继承；

3.final修饰的成员变量是常量，必须在其声明的时候赋初始值，并且赋值后不可以修改；

4.final修饰的方法无法被重写，但是可以被重载；

### 1.3.2 this和super：

1.this表示当前对象；

2.super表示当前对象的父类对象；

## 1.4 Java常用类：

### 1.4.1 object类：

#### 1.4.1.1 clone方法：

1.clone方法用于拷贝对象，这里有深度拷贝和浅度拷贝两种：

前提：使用clone方法的前提是目标类必须实现cloneable接口；

浅度拷贝：指对于对象中存在的引用类型的变量，比如String，如果执行默认的clone方法，则拷贝得到的新对象里面的string仅仅是原对象string引用的一个副本，两者都指向同一个string对象；

深度拷贝：指拷贝对象的时候不管对象里面的变量是引用类型还是基本类型，都会真正的拷贝，而不仅仅是拷贝一个引用而已；但是要实现深度拷贝，目标类必须重写父类的clone方法；

#### 1.4.1.2 getClass方法：

该方法主要利用反射来获取运行时的类类型；通常与getName方法一起使用；

#### 1.4.1.3 equal方法：

该方法默认比较两个对象的地址是否相同，和“==”一样，即比较两者是否是同一个对象，但是可以根据需求来重写equal方法，比如String类就是因为重写了equal方法，才可以比较两个string内容是不是相等；

#### 1.4.1.4 hashCode方法：

该方法用于返回对象的哈希值，一般重写equal方法的时候都会重写hashCode方法；

## 1.5 String，StringBuilder和StringBuffer

### 1.5.1 运行效率：

因为String是字符串常量，而StringBuilder和StringBuffer是字符串变量，又因为StringBuffer的方法是同步的，线程安全的，所以综上所述：这三者的运行效率如下：

StringBuilder > StringBuffer > String;

### 1.5.2 StringBuilder和StringBuffer

这两者的操作方法基本一样，只是StringBuilder是非线程安全的而已；

1. append方法：将信息追加到当前StringBuilder的结尾；
2. insert方法：将字符串或对象插入到当前StringBuilder对象的指定索引处；
3. delete方法：删除指定开始索引到结束索引处的字符；
4. length方法：输出字符串的长度；
5. reverse方法：实现字符串的反转；

## 1.6 java中修饰符的作用域以及可见性

public ：当前类、子类，同一包、其他包都可以访问；

protected：当前类，子类以及同一包可以访问，其它包不可以；

default：当前类和同一包可以访问，只要在其他包就不可以；

private：当前类可以访问，同一包、子类、其它包都不可以访问；

## 1.7 Integer和int的类型

Integer是int提供的封装类，从java5之后引入了自动装箱、拆箱机制，使得两者可以相互转换，而int是java基本数据类型；

Integer默认值是null，而int默认值是0；

Integer是对象，用一个引用指向这个对象，而int是基本类型，直接存储数据；

Integer提供了好多与整数相关的操作方法，例如：将一个字符串转换成整数等.；

Integer有缓存；

### 1.7.1 Integer缓存：

Java中的Integer对象具有缓存效果，但是仅适用于整数区间 -128 到 +127；

这种 Integer 缓存策略仅在自动装箱（autoboxing）的时候有用，使用构造器创建的 Integer 对象不能被缓存。

最大值 127 可以通过 JVM 的启动参数 -XX:AutoBoxCacheMax=size 修改（仅仅jdk6以后支持）；

缓存行为不仅适用于Integer对象。我们针对所有整数类型的类都有类似的缓存机制。

有 ByteCache 用于缓存 Byte 对象

有 ShortCache 用于缓存 Short 对象

有 LongCache 用于缓存 Long 对象

有 CharacterCache 用于缓存 Character 对象

Byte，Short，Long 有固定范围: -128 到 127。对于 Character, 范围是 0 到 127。除了 Integer 可以通过参数改变范围外，其它的都不行。

## 1.8 Java程序初始化顺序：

总结：静态字段 > 静态代码块 > 普通字段 > 构造代码块 > 构造函数

并且：父类 > 子类

1.父类静态字段初始化

2.父类静态代码块

3.子类静态字段初始化

4.子类静态代码块

5.父类普通字段初始化

6.父类构造代码块（{//代码}） –优先于构造函数执行

7.父类构造函数

8.子类普通字段初始化

9.子类构造代码块

10.子类构造函数

## 1.9 try-catch：

finally是在return后面的表达式运算之后执行的，此时并没有返回运算后的值，而是把值保存起来，不管finally对该值做了任何改变，返回的值都不会改变，依然返回保存起来的值，也就是说方法的返回值是在finally运算之前就确定了的。

如果return的数据是引用数据类型，而在finally中对该引用数据类型的属性值的改变起作用，try 中return返回的就是finally中改变后的属性值；

## 1.10 Java异常：

Java中所有的异常的超类都是Throwable；

Throwable有两个子类：有两个重要的子类：Exception（异常）和Error（错误）；

### 1.10.1 Exception（异常）：

1.程序本身可以捕获并且可以处理的异常；

2.Exception这种异常又分为两类：运行时异常和编译异常。

3.运行时异常可以捕获，也可以不捕获；而编译异常必须捕获；

### 1.10.2 Error（错误）：

1.是程序中无法处理的错误，表示运行应用程序中出现了严重的错误。此类错误一般表示代码运行时JVM出现问题。

2.这些错误是不可查的，非代码性错误。因此，当此类错误发生时，应用不应该去处理此类错误。

## 1.11 八种基础数据类型及字节大小

1.byte-------8位-------------一个字节；

2.short-------16位----------两个字节；

3.int----------32位-----------四个字节；

4.long--------64位-----------八个字节；

5.float---------32位-----------四个字节；

6.double-------64位-----------八个字节；

7.boolean-------1位；

8.char------------16位----------两个字节；

## 1.12 switch语句：

1.switch不支持long和boolean;

2.其余的基本类型以及String都支持；

## 1.13 java创建String：

1.第一种：String str = “asd”;

此时如果在常量池中不存在“asd”,则会在常量池中创建一个“asd”对象，并且返回“asd”对象的引用；

如果已经存在，则直接返回“asd”的引用，不另外创建对象；

1. String str = new String(“asd”);

此时如果常量池中不存在“asd”，则会先在常量池中创建一个“asd”对象，然后因为new关键字的作用，会继续在堆里面再创建一个“asd”对象，所以总共会创建两个对象；

如果常量池中已经存在了“asd”对象，则只是在堆里创建一个即可；

### 1.13.1 intern方法：

该方法是返回当前字符串在常量池中的引用，如果当前字符串在常量池中没有引用，则会把堆里面的引用复制到常量池中，然后再返回引用；

# 2.设计模式:

## 单例模式:

### 实现方法：

#### 饿汉式

1. 线程安全，调用效率高，但是不能延时加载：
2. 因为无法延时加载，所以即使程序运行过程没有用到该实例，这种单例模式也会初始化，这就造成了不必要的资源浪费。所以不推荐这种实现方式。
3. 实现代码如下：

public class Test2 *{* //创建私有静态变量  
 private static Test2 *instance* = new Test2*()*;  
 // 私有化构造函数，使其无法从外部实例化；  
 private Test2*(){}* //定义获取该类实例的方法  
 //因为instance是静态的，所以Test2类只会有一个实例对象  
 //从而实现了单例模式  
 public static Test2 getInstance*(){* return *instance*;  
 *}  
}*

#### 懒汉式：

1. 线程安全，调用效率不高，但是能延时加载；
2. 代码如下：

public class Test2 *{* //创建私有静态变量，但是不初始化，等到需要用的时候再初始化  
 private static Test2 *instance*;  
 // 私有化构造函数，使其无法从外部实例化；  
 private Test2*(){}* //定义获取该类实例的方法  
 //因为instance是静态的，所以Test2类只会有一个实例对象  
 //从而实现了单例模式  
 public static synchronized Test2 getInstance*(){* if*(instance* == null*){  
 instance* = new Test2*()*;  
 *}* return *instance*;  
 *}  
}*

#### 静态内部类实现模式

1. 线程安全，调用效率高，可以延时加载；
2. 代码如下：

public class Test2 *{* //构建静态内部类  
 private static class Test3*{* private static final Test2 *instance* = new Test2*()*;  
 *}* // 私有化构造函数，使其无法从外部实例化；  
 private Test2*(){}* //定义获取该类实例的方法  
 //因为instance是静态常量的，所以Test2类只会有一个实例对象  
 //从而实现了单例模式  
 public static Test2 getInstance*(){* return Test3.*instance*;  
 *}  
}*

#### 枚举实现单例模式：

1. 线程安全，调用效率高，不能延时加载，可以天然的防止反射和反序列化调用
2. 防止反射：

因为枚举在反编译之后会得到一个抽象类，而抽象类是无法实例化的，所以枚举也就是可以防止反射来实例化新的实例；

而且反编译后的抽象类中，枚举的变量的实例化代码实在静态代码块中的，所以根据jvm加载类的过程来看，枚举显然也是线程安全的；

1. 反序列化调用：

因为枚举类在jvm中的存储是唯一的，不允许重复，所以即使序列化创建新的实例，其返回的也是之前存在的实例的name；从而真正实现了单例模式；

3.代码如下：

public enum EnumTest *{* //相当于 public static final EnumTest INSTACNE;  
 *INSTACNE*;  
  
 public void way*(){* System.*out*.println*(*"使用单例模式连接数据库!!!"*)*;  
 *}  
}*

## 工厂模式：

### 简单工厂模式：

特点：

1.它是一个具体的类，非接口 抽象类。有一个重要的create()方法，利用if或者 switch创建产品并返回。

2.create()方法通常是静态的，所以也称之为静态工厂。

缺点：

1.扩展性差（我想增加一种面条，除了新增一个面条产品类，还需要修改工厂类方法）

2.当产品过多时工厂类会比较复杂；

实现代码如下：

1. 定义一个基类：

*/\*\*  
 \* 定义面条抽象类  
 \*/*public abstract class Noodle *{  
 /\*\*  
 \* 描述面条的特征的方法  
 \*/* public abstract void desc*()*;  
*}*

1. 定义各种具体的产品类：

*/\*\*  
 \* 定义兰州拉面产品类  
 \*/*public class LZNoodle extends Noodle*{* @Override  
 public void desc*() {* System.*out*.println*(*"兰州拉面"*)*;  
 *}  
}*

*\*\*  
 \* 定义泡面产品类  
 \*/*public class PaoNoodle extends Noodle*{* @Override  
 public void desc*() {* System.*out*.println*(*"统一老坛酸菜面"*)*;  
 *}  
}*

1. 创建工厂类：

*/\*\*  
 \* 定义面条工厂类  
 \*/*public class NoodleFactory *{* public static final int *TYPE\_LZ* = 1;//兰州拉面  
 public static final int *TYPE\_PAO* = 2;//泡面  
 */\*\*  
 \* 定义工厂中创建产品实例的方法  
 \** ***@param*** *type 产品类型  
 \** ***@return*** *产品实例  
 \*/* public static Noodle creatNoodle*(*int type*){* switch *(*type*){* case *TYPE\_LZ*:  
 return new LZNoodle*()*;  
 case *TYPE\_PAO*:  
 return new PaoNoodle*()*;  
 default:  
 return null;  
 *}  
 }  
}*

最后通过调用工厂类的createNoodle方法就可以得到不同的产品实例；

public static void main*(*String*[]* args*) {* NoodleFactory.*creatNoodle(*NoodleFactory.*TYPE\_LZ)*.desc*()*;  
 NoodleFactory.*creatNoodle(*NoodleFactory.*TYPE\_PAO)*.desc*()*;  
*}*

### 工厂方法模式

提供一个用于创建对象的接口(工厂接口)，让其实现类(工厂实现类)决定实例化哪一个类(产品类)，并且由该实现类创建对应类的实例。

可以一定程度上解耦，消费者和产品实现类隔离开，只依赖产品接口(抽象产品)，产品实现类如何改动与消费者完全无关。

但是用户要获取不同的产品的时候需要亲自去找不同的工厂，对于用户来讲操作太复杂；

实现代码如下：

*/\*\*  
 \* 定义一个面馆接口，功能就是做面条  
 \*/*public interface INoodle *{* Noodle getNoodle*()*;  
*}*

*/\*\*  
 \* 定义兰州拉面产品类  
 \*/*public class LZNoodle extends Noodle*{* @Override  
 public void desc*() {* System.*out*.println*(*"兰州拉面"*)*;  
 *}  
}*

*/\*\*  
 \* 定义泡面产品类  
 \*/*public class PaoNoodle extends Noodle*{* @Override  
 public void desc*() {* System.*out*.println*(*"统一老坛酸菜面"*)*;  
 *}  
}*

*/\*\*  
 \* 定义兰州拉面工厂类  
 \*/*public class LZNoodleFactory implements INoodle*{* @Override  
 public Noodle getNoodle*() {* return new LZNoodle*()*;  
 *}  
}*

*/\*\*  
 \* 定义泡面工厂类  
 \*/*public class PaoNoodleFactory implements INoodle*{* @Override  
 public Noodle getNoodle*() {* return new PaoNoodle*()*;  
 *}  
}*

调用工厂接口获取对象实例：

public static void main*(*String*[]* args*) {* INoodle lzn = new LZNoodleFactory*()*;  
 lzn.getNoodle*()*.desc*()*;  
 INoodle pn = new PaoNoodleFactory*()*;  
 pn.getNoodle*()*.desc*()*;  
*}*

### 抽象工厂模式：

优点：既可以在一定程度上解耦，也避免了用户复杂的操作；

抽象工厂模式的缺点：

产品族的扩展将是一件十分费力的事情，假如产品族中需要增加一个新的产品，则几乎所有的工厂类都需要进行修改。所以使用抽象工厂模式时，对产品等级结构的划分是非常重要的。

适用场景：

当需要创建的对象是一系列相互关联或相互依赖的产品族时，便可以使用抽象工厂模式。说的更明白一点，就是一个继承体系中，如果存在着多个等级结构（即存在着多个抽象类），并且分属各个等级结构中的实现类之间存在着一定的关联或者约束，就可以使用抽象工厂模式。假如各个等级结构中的实现类之间不存在关联或约束，则使用多个独立的工厂来对产品进行创建，则更合适一点。

实现代码如下：

*/\*\*  
 \* 定义面条抽象类  
 \*/*public abstract class Noodle *{  
  
 /\*\*  
 \* 描述面条的特征的方法  
 \*/* public abstract void desc*()*;  
*}*

*/\*\*  
 \* 定义兰州拉面产品类  
 \*/*public class LZNoodle extends Noodle*{* @Override  
 public void desc*() {* System.*out*.println*(*"兰州拉面"*)*;  
 *}  
}*

*/\*\*  
 \* 定义泡面产品类  
 \*/*public class PaoNoodle extends Noodle*{* @Override  
 public void desc*() {* System.*out*.println*(*"统一老坛酸菜面"*)*;  
 *}  
}*

*/\*\*  
 \* 定义一个面馆接口，功能就是做面条  
 \*/*public interface INoodle *{* LZNoodle creatLZNoodle*()*;  
 PaoNoodle creatPaoNoodle*()*;  
*}*

*/\*\*  
 \* 定义工厂类，实现工厂接口  
 \*/*public class Factory implements INoodle *{* @Override  
 public LZNoodle creatLZNoodle*() {* return new LZNoodle*()*;  
 *}* @Override  
 public PaoNoodle creatPaoNoodle*() {* return new PaoNoodle*()*;  
 *}  
}*

获取不同产品实例：

public static void main*(*String*[]* args*) {* INoodle factory = new Factory*()*;  
 factory.creatLZNoodle*()*.desc*()*;  
 factory.creatPaoNoodle*()*.desc*()*;  
*}*

## 2.3 观察者模式

定义了对象间一对多的依赖关系，当一个对象改变状态时，它的所有依赖者都会收到通知并自动更新。

代码实现：

1. 定义被观察者接口：

*/\*\*  
 \* 抽象被观察者接口  
 \* 定义了添加、删除、通知观察者方法  
 \*/*public interface Observerable *{  
 /\*\*  
 \* 注册观察者方法  
 \** ***@param*** *observer 观察者实例  
 \*/* void registerObserver*(*Observer observer*)*;  
 */\*\*  
 \* 删除观察者方法  
 \** ***@param*** *observer 观察者实例  
 \*/* void removeObserver*(*Observer observer*)*;  
 */\*\*  
 \* 通知观察者方法  
 \*/* void notifyObserver*()*;  
*}*

1. 定义观察者接口

*/\*\*  
 \* 定义观察者接口  
 \* 定义了一个update()方法，当被观察者调用notifyObservers()方法时，观察者的update()方法会被回调。  
 \*/*public interface Observer *{  
 /\*\*  
 \* 更新观察者方法  
 \** ***@param*** *msg 更新信息  
 \*/* public void update*(*String msg*)*;  
*}*

1. 定义被观察者实例

*/\*\*  
 \* 被观察者，也就是微信公众号服务  
 \* 实现了Observerable接口，对Observerable接口的三个方法进行了具体实现  
 \** ***@author*** *zgj  
 \*/*public class WechatServer implements Observerable*{* //存储观察者名单  
 private List*<*Observer*>* list;  
 //存储更新信息  
 private String msg;  
  
 public WechatServer*() {* list = new ArrayList*<>()*;  
 *}* @Override  
 public void registerObserver*(*Observer observer*) {* list.add*(*observer*)*;  
 *}* @Override  
 public void removeObserver*(*Observer observer*) {* if*(*!list.isEmpty*()){* list.remove*(*observer*)*;  
 *}  
 }* @Override  
 public void notifyObserver*() {* for*(*Observer observer:list*){* observer.update*(*msg*)*;  
 *}  
 }* public void setMsg*(*String msg*) {* this.msg = msg;  
 System.*out*.println*(*"微信服务更新消息："+msg*)*;  
 notifyObserver*()*;  
 *}  
}*

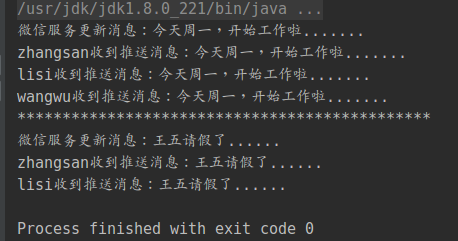
1. 定义观察者实例

*/\*\*  
 \* 定义观察者实例  
 \*/*public class User implements Observer*{* private String name;  
 private String msg;  
 public User*(*String name*) {* this.name = name;  
 *}* @Override  
 public void update*(*String msg*) {* this.msg = msg;  
 read*()*;  
 *}* public void read*() {* System.*out*.println*(*name+"收到推送消息："+msg*)*;  
 *}  
}*

1. 编写测试类

public static void main*(*String*[]* args*) {* WechatServer wechatServer = new WechatServer*()*;  
 Observer zhangsan = new User*(*"zhangsan"*)*;  
 Observer lisi = new User*(*"lisi"*)*;  
 Observer wangwu = new User*(*"wangwu"*)*;  
 wechatServer.registerObserver*(*zhangsan*)*;  
 wechatServer.registerObserver*(*lisi*)*;  
 wechatServer.registerObserver*(*wangwu*)*;  
 wechatServer.setMsg*(*"今天周一，开始工作啦......."*)*;  
 System.*out*.println*(*"\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*"*)*;  
 wechatServer.removeObserver*(*wangwu*)*;  
 wechatServer.setMsg*(*"王五请假了......"*)*;  
*}*

1. 运行结果



## 2.4 迭代器模式

迭代器模式提供一种方法顺序访问一个聚合对象中各个元素，而又不暴露该对象的内部表示。

迭代器模式分离了集合对象的遍历行为和数据存储行为，将两个行为分开，抽象出一个迭代器类来负责，这样既可以做到不暴露集合的内部结构，又可让外部代码透明地访问集合内部的数据。

优点： 1、支持不同的方式来遍历对象。 2、简化了java内部结构。 3、在一个类中可以有多个迭代器。 4、在迭代器模式中，增加新的聚合类和迭代器类都很方便，无须修改原有代码。

缺点：由于迭代器模式将存储数据和遍历数据的职责分离，增加新的聚合类需要对应增加新的迭代器类，类的个数成对增加，这在一定程度上增加了系统的复杂性。

使用场景： 1、不暴露类中的底层机构。 2、需要为类提供多种遍历方式。 3、为遍历不同的聚合结构提供一个统一的接口。

实现代码如下：

1. 创建容器类接口

*/\*\*  
 \* 定义容器类接口  
 \*/*public interface Container *{  
 /\*\*  
 \* 获取迭代器实例  
 \** ***@return*** *Iterator  
 \*/* Iterator getIterator*()*;  
*}*

1. 创建迭代器接口

*/\*\*  
 \* 定义迭代器接口  
 \*  
 \*/*public interface Iterator *{* boolean hasNext*()*;  
 Object next*()*;  
*}*

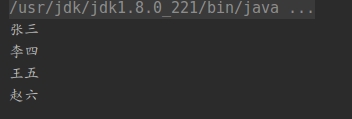
1. 创建容器实例

public class ContainerImpl implements Container*{* String*[]* names = *{*"张三","李四","王五","赵六"*}*;  
 @Override  
 public Iterator getIterator*() {* return new IteratorImpl*()*;  
 *}* private class IteratorImpl implements Iterator*{* int index = 0;  
 @Override  
 public boolean hasNext*() {* return index >=0 && index < names.length;  
 *}* @Override  
 public Object next*() {* if*(*this.hasNext*()){* return names*[*index++*]*;  
 *}* return null;  
 *}  
 }  
}*

1. 创建测试类

public static void main*(*String*[]* args*) {* Container container = new ContainerImpl*()*;  
 Iterator iterator = container.getIterator*()*;  
 while*(*iterator.hasNext*()){* System.*out*.println*(*iterator.next*())*;  
 *}  
}*

1. 运行结果



## 2.5 代理模式

代理(Proxy)是一种设计模式,提供了对目标对象另外的访问方式;即通过代理对象访问目标对象.这样做的好处是:可以在目标对象实现的基础上,增强额外的功能操作,即扩展目标对象的功能.

### 2.5.1 静态代理

1.静态代理在使用时,需要定义接口或者父类,被代理对象与代理对象一起实现相同的接口或者是继承相同父类；

2.优点：可以做到在不修改目标对象的功能前提下,对目标功能扩展.

3.缺点： 因为代理对象需要与目标对象实现一样的接口,所以会有很多代理类,类太多.同时,一旦接口增加方法,目标对象与代理对象都要维护.

4.实现代码：

创建目标类和代理类拥有共同操作的接口：

*/\*\*  
 \* 定义明星类的相关共同操作的接口  
 \*/*public interface ISuperStar *{* void work*()*;  
*}*

创建明星类，并且实现上述接口：

*/\*\*  
 \* 定义明星类  
 \*/*public class SuperStar implements ISuperStar*{* @Override  
 public void work*() {* System.*out*.println*(*"接收预约，开始参加跑男............"*)*;  
 *}  
}*

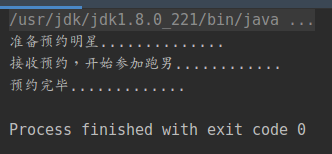
创建明星代理类，并且实现上述接口：

*/\*\*  
 \* 创建明星代理类  
 \*/*public class ProxySuperStar implements ISuperStar*{* private SuperStar superStar;  
 public ProxySuperStar*(*SuperStar superStar*){* this.superStar = superStar;  
 *}* @Override  
 public void work*() {* System.*out*.println*(*"准备预约明星.............."*)*;  
 superStar.work*()*;  
 System.*out*.println*(*"预约完毕............."*)*;  
 *}  
}*

编写测试类：

public static void main*(*String*[]* args*) {* ISuperStar proxySuperStar = new ProxySuperStar*(*new SuperStar*())*;  
 proxySuperStar.work*()*;  
*}*

运行结果：



### 2.5.2 动态代理

代理对象不需要实现接口,但是目标对象一定要实现接口,否则不能用动态代理;

代理对象的生成,是利用JDK的API,动态的在内存中构建代理对象;

实现代码如下:

明星类和明星接口代码不变；

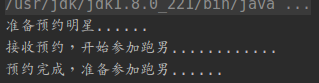
创建动态代理类：

*/\*\*  
 \* 创建明星代理类  
 \*/*public class ProxySuperStar*{* private Object target;  
 public ProxySuperStar*(*Object obj*) {* this.target = obj;  
 *}* public Object getProxyInstance*() {* return Proxy.*newProxyInstance(*target.getClass*()*.getClassLoader*()*,  
 target.getClass*()*.getInterfaces*()*,  
 new InvocationHandler*() {* @Override  
 public Object invoke*(*Object proxy, Method method, Object*[]* args*)* throws Throwable *{* System.*out*.println*(*"准备预约明星......"*)*;  
 Object returnVal = method.invoke*(*target,args*)*;  
 System.*out*.println*(*"预约完成，准备参加跑男......"*)*;  
 return returnVal;  
 *}  
 }  
 )*;  
 *}  
}*

测试类：

public static void main*(*String*[]* args*) {* ISuperStar iSuperStar = *(*ISuperStar*)*new ProxySuperStar*(*new SuperStar*())* .getProxyInstance*()*;  
 iSuperStar.work*()*;  
*}*

运行结果:



### 2.5.3 cglib代理

1.需要引入cglib的jar文件,但是Spring的核心包中已经包括了Cglib功能,所以直接引入pring-core-3.2.5.jar即可.

2.引入功能包后,就可以在内存中动态构建子类

3.代理的类不能为final,否则报错

4.目标对象的方法如果为final/static,那么就不会被拦截,即不会执行目标对象额外的业务方法.

代码如下：

/\*\* \* 目标对象,没有实现任何接口 \*/

public class UserDao {

public void save() {

System.out.println("----已经保存数据!----");

}

}

/\*\*

\* Cglib子类代理工厂

\* 对UserDao在内存中动态构建一个子类对象

\*/

public class ProxyFactory implements MethodInterceptor{

//维护目标对象

private Object target;

public ProxyFactory(Object target) {

this.target = target;

}

//给目标对象创建一个代理对象

public Object getProxyInstance(){

//1.工具类

Enhancer en = new Enhancer();

//2.设置父类

en.setSuperclass(target.getClass());

//3.设置回调函数

en.setCallback(this);

//4.创建子类(代理对象)

return en.create();

}

@Override

public Object intercept(Object obj, Method method, Object[] args, MethodProxy proxy) throws Throwable {

System.out.println("开始事务...");

//执行目标对象的方法

Object returnValue = method.invoke(target, args);

System.out.println("提交事务...");

return returnValue;

}

}

/\*\*

\* 测试类

\*/

public class App {

@Test

public void test(){

//目标对象

UserDao target = new UserDao();

//代理对象

UserDao proxy = (UserDao)new ProxyFactory(target).getProxyInstance();

//执行代理对象的方法

proxy.save();

}

}

在Spring的AOP编程中:  
如果加入容器的目标对象有实现接口,用JDK代理  
如果目标对象没有实现接口,用Cglib代理

## 2.6 适配器模式

### 2.6.1 类适配器

Target类需要way1和way2两个方法，但是这两个方法属于两个不同的类，所以通过类的继承的操作来使得其中一个源类变成适配器，得以满足target类，代码实现如下：

Target类：

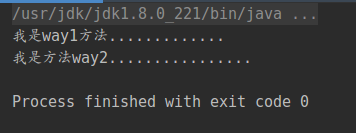
*/\*\*  
 \* 目标类，同时需要way1和way2两个方法  
 \*/*public class Target *{* public static void main*(*String*[]* args*) {* Work1 work1 = new Work1*()*;  
 work1.way1*()*;  
 work1.way2*()*;  
 *}  
}*

源：

*/\*\*  
 \* 源类2，拥有way2方法  
 \*/*public class Work2 *{* public void way2*() {* System.*out*.println*(*"我是方法way2................"*)*;  
 *}  
}*

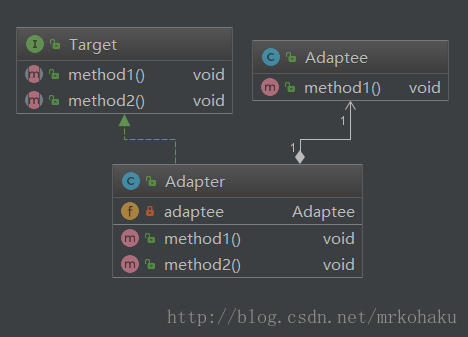
*/\*\*  
 \* 源1类，拥有way1方法  
 \* 同时将其作为适配器类，得以连接源2类  
 \*/*public class Work1 extends Work2*{* public void way1*() {* System.*out*.println*(*"我是way1方法............."*)*;  
 *}  
}*

运行结果：



### 2.6.2 对象适配器

从下面的结构图可以看出，Adaptee类并没有method2()方法，而客户端则期待这个方法。与类适配器模式一样，为使客户端能够使用Adaptee类，我们把Adaptee与Target衔接起来。但这里我们不继承Adaptee，而是把Adaptee封装进Adapter里。这里Adaptee与Adapter是组合关系;



类适配器使用的是继承的方式，直接继承了Adaptee，所以无法对Adaptee的子类进行适配。

对象适配器使用的是组合的方式，·所以Adaptee及其子孙类都可以被适配。另外，对象适配器对于增加一些新行为非常方便，而且新增加的行为同时适用于所有的源。

基于组合/聚合优于继承的原则，使用对象适配器是更好的选择。但具体问题应该具体分析，某些情况可能使用类适配器会适合，最适合的才是最好的。

### 2.6.3 接口适配器：

在任何时候，如果不准备实现一个接口里的所有方法时，就可以使用“缺省适配模式”制造一个抽象类，实现所有方法，这样，从这个抽象类再继承下去的子类就不必实现所有的方法，只要重写需要的方法就可以了。

### 2.6.4 总结

1.优点

更好的复用性：系统需要使用现有的类，而此类的接口不符合系统的需要。那么通过适配器模式就可以让这些功能得到更好的复用。

更好的扩展性：在实现适配器功能的时候，可以扩展自己源的行为（增加方法），从而自然地扩展系统的功能。

2.缺点

会导致系统紊乱：滥用适配器，会让系统变得非常零乱。例如，明明看到调用的是A接口，其实内部被适配成了B接口的实现，一个系统如果太多出现这种情况，无异于一场灾难。因此如果不是很有必要，可以不使用适配器，而是直接对系统进行重构。

# 3.Java反射：

java反射是在运行状态中，对任意一个类，都能够知道这个类的所有属性和方法，对于任何一个对象，都能够调用它的任意一个属性和方法，这种动态获取信息以及动态调用对象的方法的功能称为java的反射机制;

Class是一个类，一个描述类的类，是反射机制的基石，封装了描述方法的Method，描述字段的Filed，描述构造器的Constructor等属性，通过反射可以得到类的各个成分。

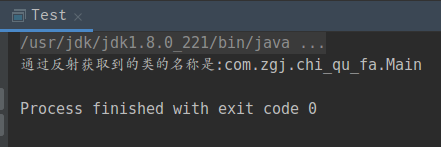
## 获取Class对象的三种方式:

### Object—getClass()； 通过已知对象获取

实现代码如下：

public class Test*{* public static void main*(*String*[]* args*) {* Main main = new Main*()*;  
 Class mc = main.getClass*()*;  
 System.*out*.println*(*"通过反射获取到的类的名称是:"+mc.getName*())*;  
 *}  
}*

执行结果如下：



### 通过类名.Class

任何数据类型（包括基本数据类型）都有一个静态的Class属性 ；

代码如下：

public class Test*{* public static void main*(*String*[]* args*) {* Class mc = Main.class;  
 System.*out*.println*(*"通过反射获取到的类的名称是:"+mc.getName*())*;  
 *}  
}*

执行结果如下：



### Class.forName(String calssName)：

代码如下：

public static void main*(*String*[]* args*){* Class mc = null;  
 try *{* mc = Class.*forName(*"com.zgj.chi\_qu\_fa.Main"*)*;  
 System.*out*.println*(*"通过反射获取到的类的名称是:"+mc.getName*())*;  
 *}* catch *(*ClassNotFoundException e*) {* e.printStackTrace*()*;  
 *}  
}*

执行结果是：



### Class.forName和classloader的区别

class.forName除了将类的.class文件加载到jvm中之外，还会对类进行解释，执行类中的static块

而classloader只做一件事情，就是将.class文件加载到jvm中，不会执行static中的内容，只有在newinstance才会去执行static块。

### 三种获取class对象的方法的区别：

1.类名.class(也称类字面常量) 方式生成Class对象不会初始化的静态块，不会初始化参数，不会调用构造函数

2. Class.forName()方式生成Class对象会初始化类静态块，但不会初始化非静态的代码块，也不调用构造函数

3 .Object.getClass()方式生成Class对象会先初始化静态块，接着执行非静态块的初始化，最后调用构造函数

## 反射中重要的类：

### 3.2.1Class类：

1. 代表一个类，位于java.lang包下。
2. 常用方法：
3. getName()：返回String形式的该类的名称。
4. newInstance() ：创建类的新实例；
5. getSuperClass()：返回某子类所对应的直接父类所对应的Class对象；
6. isArray()：判定此Class对象所对应的是否是一个数组对象；
7. getComponentType() ：如果当前类表示一个数组，则返回表示该数组组件的 Class 对象，否则返回 null。
8. getConstructor(Class[]) :返回当前 Class 对象表示的类的指定的公有构造子对象。
9. getConstructors() :返回当前 Class 对象表示的类的所有公有构造子对象数组。
10. getDeclaredConstructor(Class[]) :返回当前 Class 对象表示的类的指定已说明的一个构造子对象。
11. getDeclaredConstructors() :返回当前 Class 对象表示的类的所有已说明的构造子对象数组。
12. getDeclaredField(String) :返回当前 Class 对象表示的类或接口的指定已说明的一个域对象。
13. getDeclaredFields() :返回当前 Class 对象表示的类或接口的所有已说明的域对象数组。
14. getField(String) :返回当前 Class 对象表示的类或接口的指定的公有成员域对象。
15. getFields() :返回当前 Class 对象表示的类或接口的所有可访问的公有域对象数组。
16. getDeclaredMethod(String, Class[]) :返回当前 Class 对象表示的类或接口的指定已说明的一个方法对象。
17. getDeclaredMethods() :返回 Class 对象表示的类或接口的所有已说明的方法数组。
18. getMethod(String, Class[]) :返回当前 Class 对象表示的类或接口的指定的公有成员方法对象。（invoke方法调用该方法）
19. getMethods() :返回当前 Class 对象表示的类或接口的所有公有成员方法对象数组，包括已声明的和从父类继承的方法。

### 3.2.2 Field类：

1.代表类的成员变量（成员变量也称为类的属性）。

常用方法：

getName()：返回字段对象名称

getType():返回一个Class对象，该Class对象表示该字段对象的声明类型参数对象，会擦除泛型;

getGenericType():返回一个Type对象，该Type对象表示该字段对象的声明类型参数类型，会保留泛型;

isAccessible():获取该字段对象的可访问标志;

setAccessible(boolean flag):设置该字段对象的可访问标志;在其他类里获取该类的私有成员变量时，需要设置访问标志为true，否则会报异常;

### 3.2.3 Method类：

1.代表类的方法。

2.常用方法：

invoke()：参数为类的具体实例对象，实现对方法的动态调用；

getAnnotatedReturnType()：返回一个AnnotatedType对象，表示该方法/构造函数的返回类型；

getAnnotatedExceptionTypes()：返回一个AnnotatedType对象数组，表示该方法/构造函数声明的异常；

### 3.2.4 Constructor类：

1.代表类的构造方法。

### 3.2.5 Array类：

1.提供了动态创建数组，以及访问数组的元素的静态方法。

# 4.Java数据结构：

## 4.1 Collection

### 4.1.1 List

#### 4.1.1.1 ArrayList

1.底层是Object的数组，其数组名称是elementData，默认大小是10，可以存null；

2.arraylist对于中间元素的添加和删除速度比较慢，查询的速度快，线程不安全，效率高；（注意：如果是在尾部操作元素且不需要扩容，则其和linkedlist速度差不多）

3.每次扩容为原来的1.5倍，扩容方式是：会先创建一个原来数组1.5倍大小的新数组，然后将数据拷贝到新数组中；

4.Vector底层数据结构为数组，增删慢、查询快，线程安全，效率低，每次扩容为原来数组的2倍。

5.添加元素方法add的执行流程：

调用add()方法时，add()方法首先调用ensureCapacityInternal()来判断elementData数组容量是否足够，之所以ensureCapacityInternal()能够判断，是因为它内部调用了ensureExplicitCapacity()方法，这个方法才是真正判断elementData数组容量是否够用的方法。如果容量足够，则直接将元素添加到ArrayList中；如果容量不足，则ensureExplicityCapacity()方法内部会调用grow()方法来对数组进行扩容。扩容成功之后，再将元素添加到ArrayList扩容之后的新数组中。

1. remove方法执行流程：

首先判断index是否合理，如果合理的话，会调用System.arraycopy()方法把指定下标到数组末尾的元素向前移动一个单位，并且会把数组最后一个元素设为null。这样是为了方便GC回收。

1. get方法执行流程：

首先判断index是否合理，然后调用elementData()方法，elementData()方法返回根据index查到的具体的元素。注意：这里的返回值都经过了向下转型（Object -> E）。

1. fail\_fast机制：

Arraylist中有一个变量modCount用来存储list结构改变的次数，如果当一个线程在遍历list的时候，另外一个线程同时给list增加或删除了元素，则会导致遍历结果不正确，从而抛出异常；

1. arraylist时间复杂度：

当修改、查询或者只在数组末尾增、删时，时间复杂度为O(1)；对指定位置的元素进行增、删时，时间复杂度为O(n)。

1. arraylist遍历的三种方法：

For循环 > iterator迭代器 > 加强for循环(底层基于迭代器实现)

#### 4.1.1.2 LinkedList

1.底层是双向链表，增删快，查询慢，线程不安全，效率高；

2.add方法是采用尾插法；

3.链表结构的遍历方法：iterator迭代器最快 > 加强for循环 > for循环

4.get(i)方法实现逻辑：

先判断下标是否合法，然后根据下标的位置选择链表的遍历方向（向左，还是向右）；

### 4.1.2 Set

Set底层是基于Map实现的，所以它的初始容量是16（因为16是2的幂，能有效减少哈希碰撞）；扩容因子是0.75，扩容倍数是1；

Set 里面的元素是不能重复的；

Set 是无顺序存储的，即存储顺序和插入顺序不同；

Set不是线程安全的，解决办法：Set set = Collections.synchronizedSet(set 对象)。

常用方法：

add( ) 向集合中添加元素

clear( ) 去掉集合中所有的元素

contains( ) 判断集合中是否包含某一个元素

isEmpty( ) 判断集合是否为空

iterator( ) 主要用于递归集合，返回一个Iterator()对象

remove( ) 从集合中去掉特定的对象

size( ) 返回集合的大小

#### 4.1.2.1 HashSet

1.非线程安全，可以存储NULL；

2.底层是hashmap，哈希表算法，查询效率高，添加元素的时候底层调用的是hashmap的put方法；

3.每一个存储到 哈希 表中的对象，都得提供 hashCode() 和 equals() 方法的实现，用来判断是否是同一个对象,对于 HashSet 集合，我们要保证如果两个对象通过 equals() 方法返回 true，这两个对象的 hashCode 值也应该相同。

#### 4.1.2.2 TreeSet

1.不可重复，底层是TreeMap，使用红黑树算法，擅长于范围查询。

2.不能存储null；

3.因为TreeSet对插入的元素是有排序的，所以对于待插入的对象来讲，要实现 Comparable 接口，并要覆盖 compareTo(Object obj) 方法来自定义比较规则,如果 this > obj,返回正数 1;如果 this < obj,返回负数 -1;如果 this = obj,返回 0 ，则认为这两个对象相等。

## 4.2 Map

### 4.2.1 HashMap

1.底层是哈希表结构，哈希表又由数组+链表实现。允许null键和null值，线程不同步;

2.采用的是位运算+链地址法解决哈希地址冲突的方案；

3.链地址法：

当两个Key的hash值一致时，就需要像table数组中的Entry对象（即：链表）中追加或替换要put的节点。jdk1.7： 若需要新增节点，则向链表头部新增节点，新节点的next域指向原头部节点。jdk1.8： 若需要新增节点，则向链表尾部新增节点，原尾部节点next域指向新节点。重点来了，jdk8中在HashMap中就开始引入红黑树的数据结构，一旦链表中的节点个数超过了TREEIFY\_THRESHOLD这个阈值，就将链表转换成红黑树的结构；（TREEIFY\_THRESHOLD是树化阀值，常量值为8；UNTREEIFY\_THRESHOLD是链表化阀值，常量值为6；）

3.1 开放寻址法中最简单的是线性探测法：当碰撞发生时即一个键的散列值被另外一个键占用时，直接检查散列表中下一个位置，即将索引值加1；

4.扩容：

默认容量是16，负载因子是0.75；

扩为之前的两倍；

5.当数组长度为2的n次幂的时候，不同的key算得得index相同的几率较小，那么数据在数组上分布就比较均匀，也就是说碰撞的几率小，相对的，查询的时候就不用遍历某个位置上的链表，这样查询效率也就较高了。

6.hashmap中计算数据在数组中的位置：

1.计算key的哈希值：

static final int hash(Object key) {

int h;

return (key == null) ? 0 : (h = key.hashCode()) ^ (h >>> 16);

}

2.根据key的哈希值计算下标：

static int indexFor(int h, int length) {

return h & (length-1);

}

### 4.2.2 HashMap和HashTable：

1.HashMap是继承自AbstractMap类，而HashTable是继承自Dictionary类。不过它们都实现了同时实现了map、Cloneable（可复制）、Serializable（可序列化）这三个接口；

2.Hashtable比HashMap多提供了elments() 和contains() 两个方法：

elments() 方法继承自Hashtable的父类Dictionnary。elements() 方法用于返回此Hashtable中的value的枚举。

contains()方法判断该Hashtable是否包含传入的value。它的作用与containsValue()一致。事实上，contanisValue() 就只是调用了一下contains() 方法。

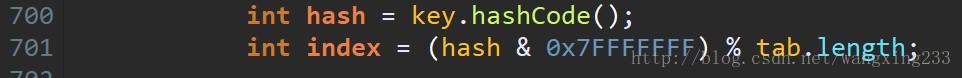
3.Hashtable既不支持Null key也不支持Null value。

4.Hashtable是线程安全的（对整个数据加锁，所以效率低下），它的每个方法中都加入了Synchronize方法，而hashmap是线程不安全的；

5.Hashtable默认的初始大小为11，之后每次扩充，容量变为原来的2n+1。HashMap默认的初始化大小为16。之后每次扩充，容量变为原来的2倍。

6.获取数组下标位置方法不同：

Hashtable直接使用对象的hashCode。hashCode是JDK根据对象的地址或者字符串或者数字算出来的int类型的数值。然后再使用除留余数发来获得最终的位置。



Hashtable在计算元素的位置时需要进行一次除法运算，而除法运算是比较耗时的。

HashMap为了提高计算效率，将哈希表的大小固定为了2的幂，这样在取模预算时，不需要做除法，只需要做位运算。位运算比除法的效率要高很多。

### 4.2.3 LinkedHashMap

继承了hashmap类，所以其主要方法都和hashmap相同；

在实现上，它底层是哈希表加链表，哈希表和hashmap一样，而链表则是用来维护其元素的插入顺序的，所以它是插入有序的，除此之外和hashmap并无太大差别；

### 4.2.4 TreeMap

1.TreeMap是一个有序的key-value集合，它是通过红黑树实现的。它的每一个元素是一个key-value对。它继承于AbstractMap，实现了NavigableMap、Cloneable、 Serializable等接口。TreeMap实现了NavigableMap接口，NavigableMap接口中定义了很多用于导航的方法，比如返回小于（大于）某个key的节点集合，返回具有最大（最小）key的节点等；实现了Cloneable接口，能被克隆；实现了Serializable接口，因此它支持序列化，能够通过序列化传输。

2.TreeMap是非线程安全的，只是用于单线程环境下。

3.ThreeMap中的几个主要的成员变量：

// 自定义的比较器，用于给TreeMap的元素排序（可以在创建ThreeMap的时候在构造函数中传入自己定义的比较器）

private final Comparator<? super K> comparator;

// 红黑树的根节点

private transient Entry<K,V> root;

// 节点数目

private transient int size = 0;

// TreeMap结构的修改次数，用于fail-fast机制

private transient int modCount = 0;

4.treemap中仅仅支持null值；

### 4.2.5 ConcurrentHashMap

是同步版本的hashmap，它底层是hashmap和一个链表，hashmap中存储数据，链表中存储锁，它采用了分段锁技术来将锁细粒度化，从而使其能达到更高的并发率，理论上其同一时间的并发数和链表的长度相等；

## 4.3 栈

1.栈是一种只能在一端进行插入或删除操作的线性表。（先进后出表）

2.常用方法：

判断是否为空：stack.empty()

取栈顶值（不出栈）：stack.peek()

进栈：stack.push(Object);

出栈：stack.pop();

1. 可应用于括号匹配的问题中；

## 4.4 堆：

1.堆分为大根堆和小根堆：

大根堆：所有节点的值均要大于它的子节点的值，小于它的父节点的值；

小根堆：所有节点的值均要小于它的子节点的值，大于它的父节点的值；

2.堆其实就是一棵二叉树，推荐用数组实现；

3.代码实现如下（小根堆）：（大根堆类似）

*/\*\*  
 \* 自定义小根堆  
 \*/*public class SmallRootHeap *{* private int*[]* array = null;  
 public SmallRootHeap*(*int size*){* array = new int*[*size*]*;  
 *}* public SmallRootHeap*(*int*[]* data*){* array = new int*[*data.length*]*;  
 System.*arraycopy(*data,0,array,0,data.length*)*;  
 *}  
  
 /\*\*  
 \* 调整小根堆的稳定性  
 \*/* public void adjust*(){* int length = array.length;  
 int fatherNodeNum = length/2;  
 //如果元素个数是偶数，则存在一个父节点只有一个子节点  
 if*(*length % 2 == 0*){* //父节点大于子节点,需要调整  
 if*(*array*[*length-1*]* < array*[*fatherNodeNum-1*]){* int temp = array*[*length-1*]*;  
 array*[*length-1*]* = array*[*fatherNodeNum-1*]*;  
 array*[*fatherNodeNum-1*]* = temp;  
 *}* fatherNodeNum--;  
 *}* //下面的父节点都同时拥有左子树和右子树  
 while*(*fatherNodeNum > 0*){* //k值一直跟随着当前节点，直到当前节点被换到叶子节点为止  
 int k = fatherNodeNum;  
 while*(*k <= length/2*){* //左孩子节点下标  
 int lnode = k\*2-1;  
 //右孩子节点下标  
 int rnode = k\*2;  
 //左子节点小于右子节点  
 if*(*array*[*lnode*]* <= array*[*rnode*]){* //左子节点小于其父节点  
 if*(*array*[*lnode*]* < array*[*k-1*]){* int temp = array*[*lnode*]*;  
 array*[*lnode*]* = array*[*k-1*]*;  
 array*[*k-1*]* = temp;  
 k = k\*2;  
 *}*else*{* break;  
 *}  
 }*else *{*//若右节点更小一些  
 //右节点小于其父节点  
 if*(*array*[*rnode*]* < array*[*k-1*]){* int temp = array*[*k-1*]*;  
 array*[*k-1*]* = array*[*rnode*]*;  
 array*[*rnode*]* = temp;  
 k = k\*2+1;  
 *}*else*{* break;  
 *}  
 }  
 }* fatherNodeNum--;  
 *}  
 }  
  
 /\*\*  
 \* 向小根堆中插入数据  
 \** ***@param*** *value  
 \** ***@return*** *\*/* public boolean insert*(*int value*){* try *{* int length = array.length+1;  
 int*[]* b = new int*[*length*]*;  
 System.*arraycopy(*array,0,b,0,length-1*)*;  
 //暂时将新元素添加到数组的最后一个位置  
 b*[*length-1*]* = value;  
 //k-1就是新加入元素的位置  
 int k = length;  
 //上限是将元素移动到根节点  
 //从下到上调整小根堆  
 while*(*k > 1*){* //新元素大于其父节点，则直接跳出循环  
 if*(*b*[*k-1*]* >= b*[*k/2-1*]){* break;  
 *}*else*{* int temp = b*[*k-1*]*;  
 b*[*k-1*]* = b*[*k/2-1*]*;  
 b*[*k/2-1*]* = temp;  
 k = k/2;  
 *}  
 }* array = b;  
 return true;  
 *}*catch *(*Exception e*){* System.*out*.println*(*"异常："*)*;  
 e.printStackTrace*()*;  
 return false;  
 *}  
 }  
  
 /\*\*  
 \* 获取并删除根节点  
 \** ***@return*** *\*/* public int poll*(){* int length = array.length-1;  
 int*[]* b = new int*[*length*]*;  
 System.*arraycopy(*array,0,b,0,length*)*;  
 //去除小根堆中最小的值，为了保持树的形状，用最后一个元素覆盖根节点  
 int result = b*[*0*]*;  
 b*[*0*]* = array*[*length*]*;  
 //从上到下调整小根堆  
 //k跟踪当前需要调整的新的根节点元素的位置  
 int k = 1;  
 while*(*k <= length/2*){* int lnode = 2\*k-1;  
 int rnode = 2\*k;  
 if*(*b*[*lnode*]* <= b*[*rnode*]){* if*(*b*[*lnode*]* < b*[*k-1*]){* int temp = b*[*k-1*]*;  
 b*[*k-1*]* = b*[*lnode*]*;  
 b*[*lnode*]* = temp;  
 k = k\*2;  
 *}*else*{* break;  
 *}  
 }*else*{* if*(*b*[*rnode*]* < b*[*k-1*]){* int temp = b*[*k-1*]*;  
 b*[*k-1*]* = b*[*rnode*]*;  
 b*[*rnode*]* = temp;  
 k = k\*2;  
 *}*else*{* break;  
 *}  
 }  
 }* array = b;  
 return result;  
 *}  
  
 /\*\*  
 \* 获取根节点（不删除）  
 \** ***@return*** *\*/* public int getRootVal*(){* return array*[*0*]*;  
 *}  
 /\*\*  
 \* 遍历最小堆  
 \*/* public void getAllValue*(){* for*(*int i:array*){* System.*out*.print*(*i+" "*)*;  
 *}* System.*out*.println*()*;  
 *}  
}*

## 4.5 队列

LinkedList类实现了Queue接口，因此我们可以把LinkedList当成Queue来用。

1.先进先出

2.常用方法：

offer（Object obj） 入队；

poll()出队；

peek()查询队首元素;

## 4.6 树

### 4.6.1 普通二叉树：

#### 4.6.1.1 二叉树性质：

(1) 在非空二叉树中，第i层的结点总数不超过 , i>=1；

(2) 深度为h的二叉树最多有 个结点(h>=1)，最少有h个结点；

(3) 对于任意一棵二叉树，如果其叶结点数为N0，而度数为2的结点总数为N2，则N0=N2+1；

(4) 具有n个结点的完全二叉树的深度为 （注：[ ]表示向下取整）

(5)有N个结点的完全二叉树各结点如果用顺序方式存储，则结点之间有如下关系：

若I为结点编号则 如果I>1，则其父结点的编号为I/2；

如果2\*I<=N，则其左孩子（即左子树的根结点）的编号为2\*I；若2\*I>N，则无左孩子；

如果2\*I+1<=N，则其右孩子的结点编号为2\*I+1；若2\*I+1>N，则无右孩子。

(6)如果二叉树具有n个节点，则其一共有C(2\*n,n)/(n+1)中形态，C是排列组合；

#### 4.6.1.2 代码实现（包括前序、中序、后序、深度优先、广度优先等遍历方式）

*/\*\*  
 \* 把一个数组的值存入二叉树中，然后进行3种方式的遍历  
 \* 采用链表存储  
 \*/*public class CommonBinTreeForLinked *{* private int*[]* array = *{*1,2,3,4,5,6,7,8,9*}*;  
 private List*<*Node*>* nodeList = null;  
 public Node getRoot*(){* if*(*nodeList.size*()* > 0*){* return nodeList.get*(*0*)*;  
 *}*else*{* return null;  
 *}  
 }  
 /\*\*  
 \* 内部类，存储树的节点  
 \*/* private static class Node*{* Node leftChild;  
 Node rightChild;  
 int data;  
  
 Node*(*int data*){* this.data = data;  
 leftChild = null;  
 rightChild = null;  
 *}  
 }  
  
 /\*\*  
 \* 基于链表建立二叉树  
 \*/* public void createBinTree*(){* nodeList = new LinkedList*<>()*;  
 //将数组中的元素转换为树的节点  
 for*(*int i = 0;i < array.length;i++*){* nodeList.add*(*new Node*(*array*[*i*]))*;  
 *}* //对前lastParentIndex-1个父节点按照父节点与孩子节点的数字关系建立二叉树  
 //因为最后一个父亲节点可能没有右孩子  
 for*(*int parentIndex = 0;parentIndex < nodeList.size*()*/2-1;parentIndex++*){* //因为下标从0开始，所以计算子节点下标的时候需要加1,而循环中parentIndex的上限要减1  
 //为当前节点引入左孩子节点  
 nodeList.get*(*parentIndex*)*.leftChild = nodeList.get*(*parentIndex\*2+1*)*;  
 //为当前节点引入右孩子节点  
 nodeList.get*(*parentIndex*)*.rightChild = nodeList.get*(*parentIndex\*2+2*)*;  
 *}* //单独处理最后一个父亲节点  
 int lastParentIndex = nodeList.size*()*/2-1;  
 //为最后一个父亲节点引入左孩子  
 nodeList.get*(*lastParentIndex*)*.leftChild = nodeList.get*(*lastParentIndex\*2+1*)*;  
 //判断整个树的节点个数，如果是奇数才会有右孩子节点  
 if*(*nodeList.size*()*%2 == 1*){* nodeList.get*(*lastParentIndex*)*.rightChild = nodeList.get*(*lastParentIndex\*2+2*)*;  
 *}* //二叉树建立完毕  
 *}  
  
 /\*\*  
 \* 先序遍历二叉树  
 \** ***@param*** *root 二叉树的根节点  
 \*/* public void preOrderTraverse*(*Node root*){* if*(*root == null*){* return;  
 *}* System.*out*.print*(*root.data+"--"*)*;  
 preOrderTraverse*(*root.leftChild*)*;  
 preOrderTraverse*(*root.rightChild*)*;  
 *}  
  
 /\*\*  
 \* 中序遍历二叉树  
 \** ***@param*** *root 二叉树的根节点  
 \*/* public void inOrderTraverse*(*Node root*){* if*(*root == null*){* return;  
 *}* inOrderTraverse*(*root.leftChild*)*;  
 System.*out*.print*(*root.data+"--"*)*;  
 inOrderTraverse*(*root.rightChild*)*;  
 *}  
  
 /\*\*  
 \* 后序遍历二叉树  
 \** ***@param*** *root 二叉树的根节点  
 \*/* public void postOrderTraverse*(*Node root*){* if*(*root == null*){* return;  
 *}* postOrderTraverse*(*root.leftChild*)*;  
 postOrderTraverse*(*root.rightChild*)*;  
 System.*out*.print*(*root.data+"--"*)*;  
 *}  
  
 /\*\*  
 \* 广度优先遍历  
 \** ***@param*** *root 二叉树根节点  
 \*/* public void breadthFirstTraverse*(*Node root*){* if*(*root == null*){* return;  
 *}* Queue*<*Node*>* queue = new LinkedList*<>()*;  
 queue.offer*(*root*)*;  
 while*(*!queue.isEmpty*()){* Node node = queue.poll*()*;  
 System.*out*.print*(*node.data+"--"*)*;  
 if*(*node.leftChild != null*){* queue.offer*(*node.leftChild*)*;  
 *}* if*(*node.rightChild != null*){* queue.offer*(*node.rightChild*)*;  
 *}  
 }  
 }  
  
 /\*\*  
 \* 深度优先遍历(非递归)  
 \** ***@param*** *root 二叉树根节点  
 \*/* public void deepFirstTraverseNotRecursion*(*Node root*){* if*(*root == null*){* return;  
 *}* Stack*<*Node*>* stack = new Stack*<>()*;  
 stack.push*(*root*)*;  
 while*(*!stack.isEmpty*()){* Node node = stack.pop*()*;  
 System.*out*.print*(*node.data+"--"*)*;  
 //因为栈的特性，所以先将右孩子入栈  
 if*(*node.rightChild != null*){* stack.push*(*node.rightChild*)*;  
 *}* if*(*node.leftChild != null*){* stack.push*(*node.leftChild*)*;  
 *}  
 }  
 }  
  
 /\*\*  
 \* 深度优先遍历(递归)  
 \** ***@param*** *root 二叉树根节点  
 \*/* public void deepFirstTraverseRecursion*(*Node root*){* if*(*root == null*){* return;  
 *}* System.*out*.print*(*root.data+"--"*)*;  
 deepFirstTraverseRecursion*(*root.leftChild*)*;  
 deepFirstTraverseRecursion*(*root.rightChild*)*;  
 *}  
  
}*

### 4.6.2 完全二叉树

#### 4.6.2.1 性质：

1.基本性质和普通二叉树相同；

2.对于一颗二叉树，假设其深度为d（d>1）。除第d层外的所有节点构成满二叉树，且第d层所有节点从左向右连续地紧密排列，这样的二叉树被称为完全二叉树；

3.代码实现和普通二叉树相同；

### 4.6.3 平衡二叉树

它是一 棵空树或它的左右两个子树的高度差的绝对值不超过1，并且左右两个子树都是一棵平衡二叉树，同时，平衡二叉树必定是二叉搜索树。

### 4.6.4 二叉查找树：

#### 4.6.4.1 性质：

若任意节点的左子树不空，则左子树上所有节点的值均小于它的根节点的值；

若任意节点的右子树不空，则右子树上所有节点的值均大于它的根节点的值；

任意节点的左、右子树也分别为二叉查找树；

没有键值相等的节点。

代码实现如下：

*/\*\*  
 \* 二叉查找树实现类  
 \*/*public class LookingBinTree *{* Node root;  
 class Node*{* int key;  
 String value;  
 Node leftChild = null;  
 Node rightChild = null;  
 @Override  
 public int hashCode*() {* return key;  
 *}* @Override  
 public boolean equals*(*Object obj*) {* return this.key == *((*Node*)*obj*)*.key;  
 *}  
 }  
  
 /\*\*  
 \* 根据键查询值  
 \** ***@param*** *key 键  
 \** ***@return*** *目标节点  
 \*/* public Node findNode*(*int key*){* Node current = root;  
 if*(*root == null*){* return null;  
 *}* while*(*current.key != key*){* if*(*key < current.key*){* //进入左子树查询  
 current = current.leftChild;  
 *}*else*{*//进入右子树查询  
 current = current.rightChild;  
 *}* if*(*current == null*){* return null;  
 *}  
 }* return current;  
 *}  
  
 /\*\*  
 \* 插入数据  
 \** ***@param*** *key 数据的键  
 \** ***@param*** *value 数据的值  
 \*/* public void insertNode*(*int key,String value*){* Node node = new Node*()*;  
 node.key = key;  
 node.value = value;  
 if*(*root == null*){* root = node;  
 return;  
 *}* //找到插入节点的位置  
 Node parent = root;  
 Node current = root;  
 while*(*true*){* parent = current;  
 if*(*key == current.key*){* //该数据已存在，执行更新操作  
 current.value = value;  
 return;  
 *}* if*(*key < current.key*){* //进入左子树查询  
 current = current.leftChild;  
 if*(*current == null*){* //已经到达叶子节点  
 parent.leftChild = node;  
 return;  
 *}  
 }*else*{* //进入右子树查询  
 current = current.rightChild;  
 if*(*current == null*){* parent.rightChild = node;  
 return;  
 *}  
 }  
 }  
 }  
  
 /\*\*  
 \* 查询待删除节点的后续节点  
 \** ***@param*** *delNode  
 \** ***@return*** *Node 返回待删除节点的后续节点  
 \*/* public Node getProcesser*(*Node delNode*){* Node parent = delNode;  
 Node success = delNode;  
 Node current = delNode.rightChild;  
 //寻找目标节点的后续节点  
 while*(*current != null*){* parent = current;  
 success = current;  
 current = current.leftChild;  
 *}* //假如我们找到的后继节点不直接是 要删除节点的右节点 而是在其右节点那条子树上面最小的一个节点  
 if*(*success != delNode.rightChild*){* //后继节点的父节点断开其与后继节点左边的引用，重新连接上后继节点的右子节点  
 // （因为后继节点是没有左子节点的，锁以要保存之前树的状态，还要把后继节点的右子节点处理一下，不管 其存在不存在）  
 parent.leftChild = success.rightChild;  
 // 这时候后继节点的右边已经空了 上一条语句已经将其给了自己父节点的左子节点  
 // 然后让后继节点的右边 连接要删除节点的右子树  
 success.rightChild = delNode.rightChild;  
 *}* return success;  
 *}  
  
 /\*\*  
 \* 删除键为key的数据  
 \** ***@param*** *key 键  
 \** ***@return*** *boolean  
 \*/* public boolean delete*(*int key*){* Node parent = root;  
 Node current = root;  
 //标记需要删除的节点是否是左节点，得以判断其父节点的修改方式（修改左节点还是右节点）  
 boolean isleftChild = true;  
 while*(*current.key != key*){* parent = current;  
 if*(*key < current.key*){* isleftChild = true;  
 current = current.leftChild;  
 *}*else*{* isleftChild = false;  
 current = current.rightChild;  
 *}* if*(*current == null*){* return false;  
 *}  
 }* //第一种情况，删除节点是子节点  
 if*(*current.leftChild == null && current.rightChild == null*){* if*(*current == root*) {* //判断待删除节点是否是根节点(此时整棵树只有一个节点)  
 root = null;  
 *}*else*{* //判断待删除节点是否是左节点  
 if*(*isleftChild*){* parent.leftChild = null;  
 *}*else*{* parent.rightChild = null;  
 *}  
 }  
 }*else if*((*current.leftChild == null && current.rightChild != null*)*||*(*current.leftChild != null && current.rightChild == null*)){* //第二种情况，删除节点只包含一个子节点，则将子节点移动动当前节点中  
 if*(*current.rightChild == null*){* if*(*current == root*){* root = current.leftChild;  
 *}*else*{* if*(*isleftChild*){* parent.leftChild = current.leftChild;  
 *}*else*{* parent.rightChild = current.leftChild;  
 *}  
 }  
 }*else*{* if*(*root == current*){* root = current.rightChild;  
 *}*else*{* if*(*isleftChild*){* parent.leftChild = current.rightChild;  
 *}*else*{* parent.rightChild = current.rightChild;  
 *}  
 }  
 }  
 }*else if*(*current.leftChild != null && current.rightChild != null*){* //第三种情况，删除节点中有左右两个节点  
 //寻找待删除节点的后续节点  
 Node processer = getProcesser*(*current*)*;  
 if*(*root == current*){* root = processer;  
 *}*else*{* if*(*isleftChild*){* parent.leftChild = processer;  
 *}*else*{* parent.rightChild = processer;  
 *}  
 }* //选中的节点的左节点与删除节点的左节点相连  
 processer.leftChild = current.leftChild;  
 *}* return true;  
 *}  
  
 /\*\*  
 \* 先序遍历二叉树  
 \** ***@param*** *root 二叉树的根节点  
 \*/* public void preOrderTraverse*(*Node root*){* if*(*root == null*){* return;  
 *}* System.*out*.println*(*root.key+"--"+root.value*)*;  
 preOrderTraverse*(*root.leftChild*)*;  
 preOrderTraverse*(*root.rightChild*)*;  
 *}  
  
 /\*\*  
 \* 中序遍历二叉树  
 \** ***@param*** *root 二叉树的根节点  
 \*/* public void inOrderTraverse*(*Node root*){* if*(*root == null*){* return;  
 *}* inOrderTraverse*(*root.leftChild*)*;  
 System.*out*.println*(*root.key+"--"+root.value*)*;  
 inOrderTraverse*(*root.rightChild*)*;  
 *}  
  
 /\*\*  
 \* 后序遍历二叉树  
 \** ***@param*** *root 二叉树的根节点  
 \*/* public void postOrderTraverse*(*Node root*){* if*(*root == null*){* return;  
 *}* postOrderTraverse*(*root.leftChild*)*;  
 postOrderTraverse*(*root.rightChild*)*;  
 System.*out*.println*(*root.key+"--"+root.value*)*;  
 *}  
  
 /\*\*  
 \* 广度优先遍历  
 \** ***@param*** *root 二叉树根节点  
 \*/* public void breadthFirstTraverse*(*Node root*){* if*(*root == null*){* return;  
 *}* Queue*<*Node*>* queue = new LinkedList*<>()*;  
 queue.offer*(*root*)*;  
 while*(*!queue.isEmpty*()){* Node node = queue.poll*()*;  
 System.*out*.println*(*root.key+"--"+root.value*)*;  
 if*(*node.leftChild != null*){* queue.offer*(*node.leftChild*)*;  
 *}* if*(*node.rightChild != null*){* queue.offer*(*node.rightChild*)*;  
 *}  
 }  
 }  
  
 /\*\*  
 \* 深度优先遍历(非递归)  
 \** ***@param*** *root 二叉树根节点  
 \*/* public void deepFirstTraverseNotRecursion*(*Node root*){* if*(*root == null*){* return;  
 *}* Stack*<*Node*>* stack = new Stack*<>()*;  
 stack.push*(*root*)*;  
 while*(*!stack.isEmpty*()){* Node node = stack.pop*()*;  
 System.*out*.println*(*root.key+"--"+root.value*)*;  
 //因为栈的特性，所以先将右孩子入栈  
 if*(*node.rightChild != null*){* stack.push*(*node.rightChild*)*;  
 *}* if*(*node.leftChild != null*){* stack.push*(*node.leftChild*)*;  
 *}  
 }  
 }  
  
 /\*\*  
 \* 深度优先遍历(递归)  
 \** ***@param*** *root 二叉树根节点  
 \*/* public void deepFirstTraverseRecursion*(*Node root*){* if*(*root == null*){* return;  
 *}* System.*out*.println*(*root.key+"--"+root.value*)*;  
 deepFirstTraverseRecursion*(*root.leftChild*)*;  
 deepFirstTraverseRecursion*(*root.rightChild*)*;  
 *}  
  
}*

### 4.6.5 B-树

1.是一种多路搜索树（并不是二叉的）

 ①任意非叶子节点最多有M个儿子；且M>2;

 ②根结点的儿子数为[2,M]；

 ③除根结点以外的非叶子结点的儿子数为[M/2, M]；

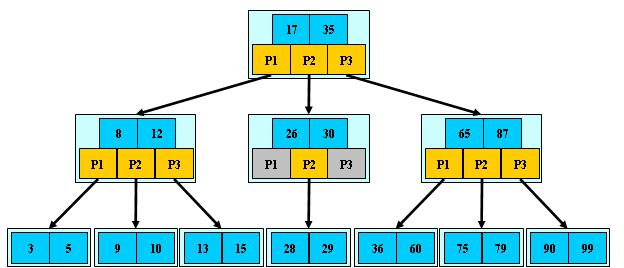
 ④每个结点存放至少M/2-1（取上整）和至多M-1个关键字；（至少2个关键字）

 ⑤非叶子结点的关键字个数=指向儿子的指针个数-1；

 ⑥非叶子结点的关键字：K[1], K[2], …, K[M-1]；且K[i] < K[i+1]；

⑦非叶子结点的指针：P[1], P[2], …, P[M]；其中P[1]指向关键字小于K[1]的子树，P[M]指向关键字大于K[M-1]的子树，其它P[i]指向关键字属于(K[i-1], K[i])的子树；

⑧所有叶子结点位于同一层；   eg:M=3



2.B-tree的特性：

①关键字集合分步在整棵树中；

②任何一个关键字出现且只出现在一个节点中；

③搜索有可能在非叶子结点结束；

④其搜索性能等价于在关键字全集内做一次二分查找；

⑤自动层次控制；

B-Tree：如果一次检索需要访问4个节点，数据库系统设计者利用磁盘预读原理，把节点的大小设计为一个页，那读取一个节点只需要一次I/O操作，完成这次检索操作，最多需要3次I/O(根节点常驻内存)。数据记录越小，每个节点存放的数据就越多，树的高度也就越小，I/O操作就少了，检索效率也就上去了。

### 4.6.6 B+树;

  1.是B-Tree的变体，也是一种多路搜索树；

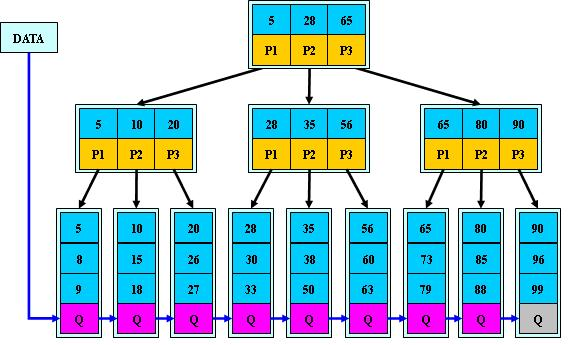
     2. 定义基本与B-Tree同，除了：

         ①非叶子结点的  子树指针=关键字数相同；

         ②非叶子结点的子树指针P[i]，指向关键字值属于 [K[i], K[i+1]) 的子树（B-树是开区间）；

         ③ 为所有叶子结点增加一个链指针；

          ④所有关键字都在叶子结点出现；  eg:（M=3）



3.B+tree树的特性：

①所有关键字都出现在叶子结点的链表中（稠密索引），且链表中的关键字恰好是有序的；

②不可能在非叶子结点命中；

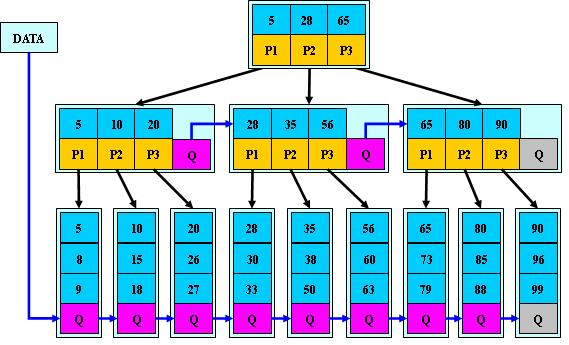
③非叶子结点相当于是叶子结点的索引（稀疏索引），叶子结点相当于存储（关键字）数据的数据层；

④更适合文件索引系统；

B+Tree：非叶子节点只存key，大大滴减少了非叶子节点的大小，那么每个节点就可以存放更多的记录，树更矮了，I/O操作更少了。所以B+Tree拥有更好的性能。

### 4.6.7 B\*树:

1. 是B+Tree的变体，在B+Tree的非根和非叶子结点再增加指向兄弟的指针；



B\*树定义了非叶子结点关键字个数至少为(2/3)\*M，即块的最低使用率为2/3

（代替B+树的1/2）；

### 4.6.7 红黑树

1.因为二叉查找树不是平衡的，有时候会出现左子树深度大大大于右子树深度，从而导致查询左子树元素接近于线性查询，效率低下，所以出现了可以自己调整平衡的红黑树；

2.红黑树基本特点和二叉查找树相同，其独有约束为：

节点是红色或者黑色；

根节点是黑色；

每个叶子节点都是黑色的空节点（null）；

每个红色节点的两个子节点都是黑色的；

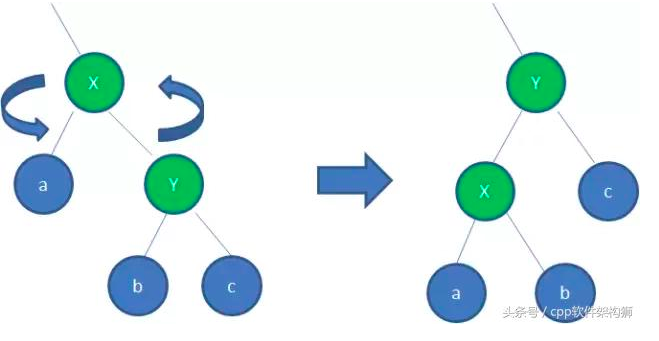
从任意节点到其每个叶子节点的所有路径都包含相同的黑色节点；

3.红黑树为了调整自身平衡的操作：

1.变色；（改变目标节点的原有颜色）

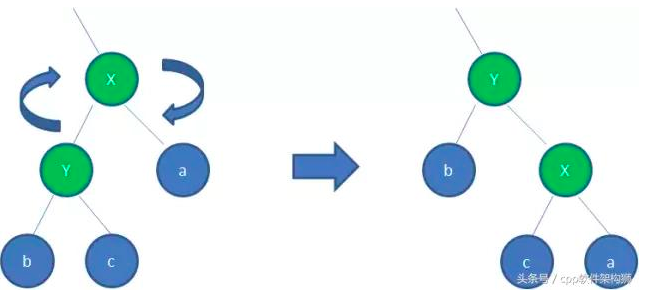
2.左旋转：

逆时针旋转节点，使得父节点被自己的右孩子节点取代，而自己成为自己的左孩子：如下图



3.右旋转：

顺时针旋转节点，使得父节点被自己的左孩子节点取代，而自己成为自己的右孩子；如下图：



## 4.6 图

# 5. 常见算法：

## 5.1 排序算法

### 5.1.1 稳定性算法：

#### 5.1.1.1 插入排序----直接插入：

数据量不大，对算法的稳定性有要求，且数据局部或者整体有序的情况。

直接插入排序法的排序原则是：将一组无序的数字排列成一排，左端第一个数字为已经完成排序的数字，其他数字为未排序的数字。然后从左到右依次将未排序的数字插入到已排序的数字中。

#### 5.1.1.2 交换排序----冒泡排序

适用于数据量量不大，对稳定性有要求，且数据基本有序的情况下

时间复杂度是O(n^2)

#### 5.1.1.3 归并排序

把序列分成元素尽可能相等的两半。

把两半元素分别进行排序。

把两个有序表合并成一个。

### 5.1.2 非稳定性算法

#### 5.1.2.1 插入排序---shell排序

希尔排序又称“缩小增量排序”，该方法的基本思想是：先将整个待排元素序列分割成若干个子序列（由相隔某 个“增量”的元素组成的）分别进行直接插入排序，然后依次缩减增量再进行排序，待整个序列中的元素基本有序（增量足够小）时，再对全体元素进行一次直接插 入排序。因为直接插入排序在元素基本有序的情况下（接近最好情况），效率是很高的，因此希尔排序在时间效率上比前两种方法有较大提高。

时间复杂度：O(n^2)

#### 5.1.2.2 选择排序----直接选择

数据量不大，且对稳定性没有要求的时候，适用于选择排序

选择排序的第一趟处理是从数据序列所有n个数据中选择一个最小的数据作为有序序列中的第1个元素并将它定位在第一号存储位置，第二趟处理从数据序列的n-1个数据中选择一个第二小的元素作为有序序列中的第2个元素并将它定位在第二号存储位置，依此类推，当第n-1趟处理从数据序列的剩下的2个元素中选择一个较小的元素作为有序序列中的最后第2个元素并将它定位在倒数第二号存储位置，至此，整个的排序处理过程就已完成。

#### 5.1.2.3 选择排序---堆排序

时间复杂度：O(n\*logn)

#### 5.1.2.4 交换排序----快速排序

快速排序（Quicksort）是对冒泡排序的一种改进。由C. A. R. Hoare在1962年提出。它的基本思想是：通过一趟排序将要排序的数据分割成独立的两部分，其中一部分的所有数据都比另外一部分的所有数据都要小，然 后再按此方法对这两部分数据分别进行快速排序，整个排序过程可以递归进行，以此达到整个数据变成有序序列。

时间复杂度：O(n\*logn)

## 5.2 查找算法

查找算法分类：

　　1）静态查找和动态查找；

　　　　注：静态或者动态都是针对查找表而言的。动态表指查找表中有删除和插入操作的表。

　　2）无序查找和有序查找。

　　　　无序查找：被查找数列有序无序均可；

　　　　有序查找：被查找数列必须为有序数列。

### 5.2.1 顺序查找

　　说明：顺序查找适合于存储结构为顺序存储或链接存储的线性表。

基本思想：顺序查找也称为线形查找，属于无序查找算法。从数据结构线形表的一端开始，顺序扫描，依次将扫描到的结点关键字与给定值k相比较，若相等则表示查找成功；若扫描结束仍没有找到关键字等于k的结点，表示查找失败。

### 5.2.2 二分查找

　　说明：元素必须是有序的，如果是无序的则要先进行排序操作。

基本思想：也称为是折半查找，属于有序查找算法。用给定值k先与中间结点的关键字比较，中间结点把线形表分成两个子表，若相等则查找成功；若不相等，再根据k与该中间结点关键字的比较结果确定下一步查找哪个子表，这样递归进行，直到查找到或查找结束发现表中没有这样的结点。

### 5.2.3 分块查找

　　分块查找又称索引顺序查找，它是顺序查找的一种改进方法。

　　算法思想：将n个数据元素"按块有序"划分为m块（m ≤ n）。每一块中的结点不必有序，但块与块之间必须"按块有序"；即第1块中任一元素的关键字都必须小于第2块中任一元素的关键字；而第2块中任一元素又都必须小于第3块中的任一元素，……

　　算法流程：

　　step1 先选取各块中的最大关键字构成一个索引表；

step2 查找分两个部分：先对索引表进行二分查找或顺序查找，以确定待查记录在哪一块中；然后，在已确定的块中用顺序法进行查找。

## 5.3 背包问题

## 5.4 尺取法

## 5.5 加密算法

## 5.6 启发式算法（上海电气）

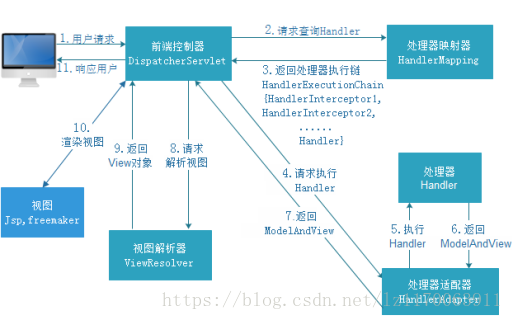
## 5.7 最短寻址算法

最短寻道时间优先算法(SSTF)：从等待的访问者中挑选寻找时间最短的那个请求执行。

# 6. web开发服务框架

## 6.1 springMVC

### 6.1.1 springMVC工作流程



1、 用户发送请求至前端控制器DispatcherServlet

2、 DispatcherServlet收到请求调用HandlerMapping处理器映射器。

3、 处理器映射器根据请求url找到具体的处理器，生成处理器对象及处理器拦截器(如果有则生成)一并返回给DispatcherServlet。

4、 DispatcherServlet通过HandlerAdapter处理器适配器调用处理器

5、 执行处理器(Controller，也叫后端控制器)。

6、 Controller执行完成返回ModelAndView

7、 HandlerAdapter将controller执行结果ModelAndView返回给DispatcherServlet

8、 DispatcherServlet将ModelAndView传给ViewReslover视图解析器

9、 ViewReslover解析后返回具体View

10、 DispatcherServlet对View进行渲染视图（即将模型数据填充至视图中）。

11、 DispatcherServlet响应用户

### 6.1.2 springmvc与struts2不同

1）springmvc的入口是一个DispatcherServlet前端控制器，而struts2入口是一个filter过滤器。

2）springmvc是基于方法开发，请求参数绑定在方法的形参上，一般设计为单例，struts2是基于类开发，请求参数绑定在类的成员变量上，只能设计为多例。

3）SpringMVC将数据存放在request域中，Struts2将数据存放在值栈中。

### 6.1.3 springMVC的配置文件通常起名叫什么？

springmvc.xml

### 6.1.4 如何在web.xml中配置springMVC的前端控制器DispatchServlet？

<!-- springmvc的前端控制器 -->

<servlet>

<servlet-name>e3-manager</servlet-name>

<servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>

<!-- contextConfigLocation不是必须的， 如果不配置contextConfigLocation， springmvc的配置文件默认在：WEB-INF/servlet的name+"-servlet.xml" -->

<init-param>

<param-name>contextConfigLocation</param-name>

<param-value>classpath:spring/springmvc.xml</param-value>

</init-param>

<load-on-startup>1</load-on-startup>

</servlet>

<servlet-mapping>

<servlet-name>e3-manager</servlet-name>

<url-pattern>/</url-pattern>

</servlet-mapping>

### 6.1.5 springMVC的中心指的是什么？

前端控制器DispatcherServlet。

### 6.1.6 springMVC的三大组件指的是什么？

处理器映射器HandlerMapping、处理器适配器HandlerAdapter、视图解析器ViewResolver。

### 6.1.7 配置springMVC的时候，一般会配哪些东西呢？

在springMVC的主配置文件中配置Controller扫描包，配置springMVC的三大组件，并且在web.xml中配置前端控制器。

<!-- 配置controller扫描包 -->

<context:component-scan base-package="cn.e3mall.controller" />

<!-- 注解驱动(配置处理器映射器和处理器适配器) -->

<mvc:annotation-driven />

<!-- 配置视图解析器 -->

<bean

class="org.springframework.web.servlet.view.InternalResourceViewResolver">

<!-- 配置逻辑视图的前缀 -->

<property name="prefix" value="/WEB-INF/jsp/" />

<!---配置逻辑视图的后缀 -->

<property name="suffix" value=".jsp" />

</bean>

### 6.1.8 在发送Post请求时，经常会出现乱码的问题，你是如何解决的？

在web.xml中配置一个过滤器。

<!-- 解决post乱码 -->

<filter>

<filter-name>CharacterEncodingFilter</filter-name>

<filter-class>org.springframework.web.filter.CharacterEncodingFilter</filter-class>

<init-param>

<param-name>encoding</param-name>

<param-value>utf-8</param-value>

</init-param>

</filter>

<filter-mapping>

<filter-name>CharacterEncodingFilter</filter-name>

<url-pattern>/\*</url-pattern>

</filter-mapping>

### 6.1.8 在发送Get请求时，经常会出现乱码的问题，你是如何解决的？

两种办法：（1）修改Tomcat配置文件，添加编码与工程编码一致；

（2）对参数进行重新编码，ISO8859-1是Tomcat默认编码，需要将Tomcat编码后的内容按utf-8编码。

在实际开发中，一般不会遇到Get请求中文乱码的问题，因为我们的原则是能用Post发送请求，不用Get。

### 6.1.9 在class上添加RequestMapping(url)有什么用？

限制此类下的所有方法请求url必须以请求前缀开头。 （统一请求父路径）

### 6.1.10 @ResponseBody注解的作用是什么？

@ResponseBody注解实现将Controller方法返回的java对象转换为JSON数据格式响应给客户端。

### 6.1.11 如何在springMVC中配置拦截器？

在springMVC中使用拦截器，首先要自定义一个类实现HandlerInterceptor接口，然后实现以下方法：

 preHandle():执行handler之前执行此方法

 postHandle():handler执行之后，返回ModelAndView之前

  afterHandle():完成处理，返回ModelAndView之后

最后在springmvc.xml中配置拦截器：

<!-- 配置拦截器 -->

<mvc:interceptors>

<mvc:interceptor>

<!-- 所有的请求都进入拦截器 -->

<mvc:mapping path="/\*\*" />

<!-- 配置具体的拦截器 -->

<bean class="cn.itcast.ssm.interceptor.HandlerInterceptor1" />

</mvc:interceptor>

<mvc:interceptor>

<!-- 所有的请求都进入拦截器 -->

<mvc:mapping path="/\*\*" />

<!-- 配置具体的拦截器 -->

<bean class="cn.itcast.ssm.interceptor.HandlerInterceptor2" />

</mvc:interceptor>

</mvc:interceptors>

### 6.1.12 如果在拦截请求时，我想拦截提交参数中包含“type=test”的字符串，怎么配置？

可以在@RequestMapping注解里面加上params=”type=test”。

### 6.1.13 springMVC中方法的返回值有哪些？

有三种。分别是ModelAndView、void和String。

### 6.1.14 springMVC怎样设定重定向和转发的？

在springmvc中，我们可以这样进行转发：

ModelAndView mv=new ModelAndView("success");

mv.addObject("name", "爱琴孩");

return mv;

或者：

//保存数据

model.addAttribute("name", "爱琴孩");

//返回对应视图

return "success";

重定向：

1. 无参重定向：

return "redirect:/redirectController/redirectTest";

1. 有参数重定向：

return "redirect:/redirectController/redirectTest?name"+"aiqinhai"

注意：这种方式会出现中文乱码，并且暴露参数到服务器链接，所以安全性低，不推荐使用；

1. 有参数重定向：

@Controller

@RequestMapping(value="/TestController")

public class TestController extends BaseController{

@RequestMapping(value="/test")

public String test(Model model,RedirectAttributes attributes){

attributes.addAttribute("name", "爱琴孩");

return "redirect:/redirectController/redirectTest";

}

}

目标类：

/\*\*

\* @ClassName: RedirectController

\* @Description: TODO(测试重定向之后的Controller)

\* @author 爱琴孩

\*/

@Controller

@RequestMapping("/redirectController")

public class RedirectController {

@RequestMapping("/redirectTest")

public String redirectTest(HttpServletRequest httpServletRequest,HttpServletResponse

response,@ModelAttribute("name") String name){

System.out.println("跳转成功！恭喜你"+name);

return "success";

}

}

### 6.1.15 重定向和转发的区别是什么？

重定向：以前的request中存放的变量全部失效，并进入一个新的request作用域。

转发：以前的request中存放的变量不会失效，就像把两个页面拼到了一起。

### 6.1.16 springMVC用什么对象从后台向前台传递数据的？

可以通过Model对象，用里面的addAttribute()方法将数据加到对象里面，前台就可以通过EL表达式拿到。

### 6.1.17 @RequestBody和@ResponseBody有什么区别?

@RequestBody注解实现接收http请求的json数据，将json数据转换为java对象进行绑定；@ResponseBody注解实现将Controller方法返回的java对象转换为json响应给客户端。

## 6.2 springboot

### 6.2.1 什么是 Spring Boot？

Spring Boot 是 Spring 开源组织下的子项目，是 Spring 组件一站式解决方案，主要是简化了使用 Spring 的难度，简省了繁重的配置，提供了各种启动器，开发者能快速上手。

### 6.2.2 Spring Boot 优点

独立运行、简化配置、上手快；

### 6.2.3 Spring Boot 的核心配置文件有哪几个？它们的区别是什么？

Spring Boot 的核心配置文件是 application 和 bootstrap 配置文件。

application 配置文件，主要用于 Spring Boot 项目的自动化配置。

bootstrap 配置文件有以下几个应用场景：

使用 Spring Cloud Config 配置中心时，这时需要在 bootstrap 配置文件中添加连接到配置中心的配置属性来加载外部配置中心的配置信息；

一些固定的不能被覆盖的属性；

一些加密/解密的场景；

### 6.2.4 Spring Boot 的配置文件有哪几种格式？它们有什么区别？

.properties 和 .yml：

它们的区别主要是书写格式不同。

.yml 格式不支持 @PropertySource 注解导入配置。

### 6.2.5 Spring Boot 的核心注解是哪个？它主要由哪几个注解组成的？

启动类上面的注解是@SpringBootApplication，它也是 Spring Boot 的核心注解，主要组合包含了以下 3 个注解：

@SpringBootConfiguration：组合了 @Configuration 注解，实现配置文件的功能。

@EnableAutoConfiguration：打开自动配置的功能，也可以关闭某个自动配置的选项，如关闭数据源自动配置功能：@SpringBootApplication(exclude = { DataSourceAutoConfiguration.class })。

@ComponentScan：Spring组件扫描。

### 6.2.6 如何理解 Spring Boot 中的 Starters？

Starters可以理解为启动器，它包含了一系列可以集成到应用里面的依赖包，你可以一站式集成 Spring 及其他技术，而不需要到处找示例代码和依赖包。如你想使用 Spring JPA 访问数据库，只要加入 spring-boot-starter-data-jpa 启动器依赖就能使用了。

Starters包含了许多项目中需要用到的依赖，它们能快速持续的运行，都是一系列得到支持的管理传递性依赖。

### 6.2.7 如何在 Spring Boot 启动的时候运行一些特定的代码？

可以实现接口 ApplicationRunner 或者 CommandLineRunner，这两个接口实现方式一样，它们都只提供了一个 run 方法；

### 6.2.8 Spring Boot 有哪几种读取配置的方式？

假定配置文件如下：

com.xiaohang.controller.pet.no = ${random.uuid}com.xiaohang.controller.pet.name = Tom

Spring Boot 可以通过 ：

1.@PropertySource,

也是加在类上面的，不过可以指定加载的配置文件：

@PropertySource(value = "classpath:bean/pet.properties")

2.@Value,

@**Value**(**value** = "${com.xiaohang.controller.pet.name}")

private String name;

3.@ConfigurationProperties

先引入依赖：

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-configuration-processor</artifactId>

<optional>true</optional> </dependency>

然后创建类：

@**ConfigurationProperties**(**prefix** = "com.xiaohang.controller.pet")

@Data

public class PetBean {

**private** **String** **no**;

**private** **String** **name**;

}

最后需要在启动类上加：

@**EnableConfigurationProperties**({**PetBean**.class})

来绑定变量

### 6.2.9 SpringBoot 实现热部署有哪几种方式？

Spring Loaded

Spring-boot-devtools

### 6.2.10 Spring Boot 可以兼容老 Spring 项目吗，如何做？

可以兼容，使用 @ImportResource 注解导入老 Spring 项目配置文件。

### 6.2.11 springboot中常用的starter的组件有哪些.

spring-boot-starter-parent //boot项目继承的父项目模块.

spring-boot-starter-web //boot项目集成web开发模块.

spring-boot-starter-tomcat //boot项目集成tomcat内嵌服务器.

spring-boot-starter-test //boot项目集成测试模块.

mybatis-spring-boot-starter //boot项目集成mybatis框架.

### 6.2.12 springboot支持的默认日志框架有哪些.可以进行哪些设置.

[spring-boot: 默认采用Logback作为日志框架.  
配置即可:  
logging.file=d:/test/log.log  
logging.level.org.springframework.web=DEBUG](mailto:br/>@Value()

## 6.3 springcoud

### 6.4.1 事务机制

#### 6.4.1.1 事务：

原子性（Atomicity）：事务是一个原子操作，由一系列动作组成。事务的原子性确保动作要么全部完成，要么完全不起作用。

一致性（Consistency）：一旦事务完成（不管成功还是失败），系统必须确保它所建模的业务处于一致的状态，而不会是部分完成部分失败。在现实中的数据不应该被破坏。

隔离性（Isolation）：可能有许多事务会同时处理相同的数据，因此每个事务都应该与其他事务隔离开来，防止数据损坏。

持久性（Durability）：一旦事务完成，无论发生什么系统错误，它的结果都不应该受到影响，这样就能从任何系统崩溃中恢复过来。通常情况下，事务的结果被写到持久化存储器中。

#### 6.4.1.2 传播机制

PROPAGATION\_REQUIRED：

表示当前方法必须运行在事务中。如果当前事务存在，方法将会在该事务中运行。否则，会启动一个新的事务；

PROPAGATION\_SUPPORTS：

表示当前方法不需要事务上下文，但是如果存在当前事务的话，那么该方法会在这个事务中运行；

PROPAGATION\_MANDATORY：

表示该方法必须在事务中运行，如果当前事务不存在，则会抛出一个异常；PROPAGATION\_REQUIRED\_NEW：

表示当前方法必须运行在它自己的事务中。如果存在当前事务，在该方法执行期间，当前事务会被挂起。

PROPAGATION\_NOT\_SUPPORTED：

表示该方法不应该运行在事务中。如果存在当前事务，在该方法运行期间，当前事务将被挂起。

PROPAGATION\_NEVER：

表示当前方法不应该运行在事务上下文中。如果当前正有一个事务在运行，则会抛出异常；

PROPAGATION\_NESTED：

表示如果当前已经存在一个事务，那么该方法将会在嵌套事务中运行。嵌套的事务可以独立于当前事务进行单独地提交或回滚。如果当前事务不存在，那么其行为与PROPAGATION\_REQUIRED一样。

## 6.4 spring

### 6.4.2 AOP的应用场景、Aop原理、好处？

面向切面编程；用来封装横切关注点，具体可以在下面的场景中使用:

Authentication 权限、Caching 缓存、Context passing 内容传递、Transactions 事务

原理：AOP是面向切面编程，是通过动态代理的方式为程序添加统一功能，集中解决一些公共问题。

优点：1.各个步骤之间的良好隔离性耦合性大大降低

1. 源代码无关性，再扩展功能的同时不对源码进行修改操作

### 6.4.3 Spring中IOC的作用与原理？对象创建的过程。

控制反转。当某个角色需要另外一个角色协助的时候，在传统的程序设计过程中，通常由调用者来创建被调用者的实例对象。但在spring中创建被调用者的工作不再由调用者来完成，因此称为控制反转。创建被调用者的工作由spring来完成，然后注入调用者 直接使用。

### 6.4.4 Spring常见创建对象的注解？

@Component@Controller@ Service@ Repository

### 6.4.5 Spring中用到的设计模式

简单工厂、工厂方法、单例模式、适配器、包装器、代理、观察者、策略、模板方法

### 6.4.6 Spring Bean的作用域之间有什么区别？

Spring容器中的bean可以分为4个范围：

singleton：这种bean范围是默认的，这种范围确保不管接受到多少个请求，每个容器中只有一个bean的实例，单例的模式由bean factory自身来维护。

prototype：原形范围与单例范围相反，为每一个bean请求提供一个实例。

request：在请求bean范围内会为每一个来自客户端的网络请求创建一个实例，在请求完成以后，bean会失效并被垃圾回收器回收。

Session：与请求范围类似，确保每个session中有一个bean的实例，在session过期后，bean会随之失效。

### 6.4.7 spring中自动装配的方式有哪些？

1.byName：

通过属性的名字的方式查找JavaBean依赖的对象并为其注入。比如说类Computer有个属性printer，指定其autowire属性为byName后，Spring IoC容器会在配置文件中查找id/name属性为printer的bean，然后使用Seter方法为其注入。

2.byType：

通过属性的类型查找JavaBean依赖的对象并为其注入。比如类Computer有个属性printer，类型为Printer，那么，指定其autowire属性为byType后，Spring IoC容器会查找Class属性为Printer的bean，使用Seter方法为其注入。

3.constructor：

通byType一样，也是通过类型查找依赖对象。与byType的区别在于它不是使用Seter方法注入，而是使用构造子注入。

4.autodetect：

在byType和constructor之间自动的选择注入方式。

### 6.4.8 spring中的核心类有那些，各有什么作用？

BeanFactory：产生一个新的实例，可以实现单例模式

BeanWrapper：提供统一的get及set方法

ApplicationContext:提供框架的实现，包括BeanFactory的所有功能

## 6.5Netty （网易游戏）

## 6.6Kafka（网易游戏）

## 6.7 ibatis

## 6.8 struts

## 6.9 struts2

# 7.数据库相关：

## 7.1数据库mysql

### 7.1.1 常用的数据引擎：

#### 7.1.1.1 ISAM:

ISAM执行读取操作的速度很快，而且不占用大量的内存和存储资源。ISAM的两个主要不足之处在于，它不 支持事务处理，也不能够容错：如果你的硬盘崩溃了，那么数据文件就无法恢复了。

#### 7.1.1.2 MYISAM:

MyISAM使用一种表格锁定的机制，来优化多个并发的读写操作，其代价是你需要经常运行OPTIMIZE TABLE命令（整理删除操作后在数据文件中留下碎片空间，只对MyISAM, BDB和InnoDB表起作用），来恢复被更新机制所浪费的空间。MyISAM还有一些有用的扩展，例如用来修复数据库文件的MyISAMCHK工具和用来恢复浪费空间的 MyISAMPACK工具。MYISAM强调了快速读取操作；MyISAM格式的一个重要缺陷就是不能在表损坏后恢复数据。

MyISAM中有两种索引，分别是主索引和辅助索引，在这里面的主索引使用具有唯一性的键值进行创建，而辅助索引中键值可以是相同的。MyISAM分别会存一个索引文件和数据文件。它的主索引是非聚集索引。当我们查询的时候我们找到叶子节点中保存的地址，然后通过地址我们找到所对应的信息。

#### 7.1.1.3 HEAP：

HEAP允许只驻留在内存里的临时表格。驻留在内存里让HEAP要比ISAM和MYISAM都快，但是它所管理的数据是不稳定的，而且如果在关机之前没有进行保存，那么所有的数据都会丢失。在数据行被删除的时候，HEAP也不会浪费大量的空间。HEAP表格在你需要使用SELECT表达式来选择和操控数据的时候非常有用。要记住，在用完表格之后就删除表格。

#### 7.1.1.4 InnoDB：

ISAM和MyISAM数据库引擎不支持事务处理（transaction process）也不支持外来键。尽管要比ISAM和 MyISAM引擎慢很多，但是InnoDB包括了对事务处理和外来键的支持，这两点都是前两个引擎所没有的。

InnoDB索引和MyISAM最大的区别是它只有一个数据文件，在InnoDB中，表数据文件本身就是按B+Tree组织的一个索引结构，这棵树的叶节点数据域保存了完整的数据记录。所以我们又把它的主索引叫做聚集索引。而它的辅助索引和MyISAM也会有所不同，它的辅助索引都是将主键作为数据域。所以，这样当我们查找的时候通过辅助索引要先找到主键，然后通过主索引再找到对于的主键，得到信息。

MyISAM表索引在处理文本索引时更具优势，而INNODB表索引在其它类型上更具效率优势，同时MySQL高并发需要事务场景时，只能使用INNODB表;

### 7.1.3 常用指令：

#### 7.1.3.1 增删改查略过；

1. UNION 操作符用于连接两个以上的 SELECT 语句的结果组合到一个结果集合中。多个 SELECT 语句会删除重复的数据。比如：

SELECT expression1, expression2, ... expression\_n

FROM tables

[WHERE conditions]

UNION [ALL | DISTINCT]

SELECT expression1, expression2, ... expression\_n

FROM tables

[WHERE conditions];

2.WITH ROLLUP 可以实现在分组统计数据基础上再进行相同的统计（SUM,AVG,COUNT…）。

比如：

SELECT name, SUM(singin) as singin\_count FROM employee\_tbl GROUP BY name WITH ROLLUP;

3. coalesce 来设置一个可以取代 NUll 的名称，coalesce 语法：

coalesce(a,b,c);

参数说明：如果a==null,则选择b；如果b==null,则选择c；如果a!=null,则选择a；如果a b c 都为null ，则返回为null（没意义）。

4.事务：

用 BEGIN, ROLLBACK, COMMIT来实现：

BEGIN 开始一个事务

ROLLBACK 事务回滚

COMMIT 事务确认

直接用 SET 来改变 MySQL 的自动提交模式:

SET AUTOCOMMIT=0 禁止自动提交

SET AUTOCOMMIT=1 开启自动提交

#### 7.1.3.2 约束：

1.添加主键：

ALTER TABLE table\_name ADD CONSTRAINT pk\_name PRIMARY KEY(列名);

alter table tb\_name add primary key (字段1,字段2,字段3);

删除表的主键：alter table 表名 drop primary key;

2.添加外键：

ALTER TABLE table\_name ADD FOREIGN KEY (info\_id)[子表列] REFERENCES target\_table[主表名] (id)[主表列] ;

解除外键约束：alter table vip drop foreign key FK1C81D1738DA76（这里是外键字段的另外一个值）；

3.添加唯一性约束

ALTER TABLE `table\_name` ADD unique(name);

4.创建索引

CREATE INDEX 索引名字 ON 表名(字段名)

#### 7.1.3.2 登录MySQL:

1.远端登录：

mysql -h 127.0.0.1 -u root -p -P 3306

2.本地登录：

mysql -u root -p  -P 3306

#### 7.1.3.3 MySQL用户信息：

1.用户信息存储在mysql库下的user中；可以对其中的用户删改查(添加建议使用专有指令操作)；

2.添加用户：

create user [用户名] identified by '[密码]';

create user vip identified by 'vippp'; -- 所有连接

create user vip@'127.0.0.1' identified by 'xxx'; -- 本地连接

create user vip@'192.168.%' identified by 'yyy'; -- 192.168 网段的连接

3.查看当前用户：

select user() ,current\_user();

4.修改密码：

mysqladmin -u “用户名” -p password “新密码”；（不用登陆,但是回车后需要输入原密码）

5.查看用户权限

show grants for “用户名”;

6.赋予用户权限

grant select on test.\* to zx\_root;(给zx\_root用户赋予test库中所有表的查询权限)

grant select，update，delete ，insert on dmc\_db.\* to zx\_root;

7.回收权限：

revoke select on test.\* from zx\_root; //如果权限不存在会报错

8.grant和revoke可以在几个层次上控制访问权限

整个服务器，使用 grant ALL  和revoke  ALL

整个数据库，使用on  database.\*

特点表，使用on  database.table

9.刷新：

flush privileges ;

10.删除用户

drop user ‘用户名’;（或者直接从user库中删除）

delete from mysql.user where user='用户名' and host='%'

#### 7.1.3.4 字段：

添加字段：

alter table 表名 add 字段名 类型 ;

修改表中的字段为空：

alter table 表名 modify 字段名 类型 null

修改表中的字段不为空：

alter table 表名 modify 字段名 类型 not null

删除表的字段：

alter table 表名 drop column 字段;

#### 7.1.3.5 查看表的结构：

desc table\_name;

查看数据库表创建的sql语句：show create table table\_name;

#### 7.1.3.6 对sql文件的操作：

1.从mysql中导出数据库test：

在终端运行：mysqldump -h localhost -u root -p test > /home/chuzj/daochu.sql

其中localhost是服务器名字 test是数据库名字，其后可以直接跟表名（则会导出该表的数据）；

2.将daochu.sql 导入到mysql数据库：

在终端运行：mysql -h localhost -u root -p test < /home/chuzj/daochu.sql --default-character-set=utf8

其中：--default-character-set=utf8 解决中文乱码

导入mysql方法2(测试好用)  
 进入linux命令命令行下：  
 mysql -u root -p 回车 输入密码  
 mysql> use weifang  
 mysql> source /home/user/data/fileName.sql

3.导出整个数据库中的所有的表结构

在linux命令行下输入：

mysqldump -u userName -p -d dabaseName [tableName]> fileName.sql

#### 7.1.3.7 操作引擎：

1.查询某张表格的引擎：

show create table tableName;

1. 改变原有表格引擎：

ALTER TABLE tableName CHANGE TYPE=InnoDB;

(MySQL用三步来实现这一目的。首先，这个表格的一个副本被创建。然后，任何输入数据的改变都被排入队列，同时这个副本被移动到另一个引擎。最后，任何排入队列的数据改变都被送交到新的表格里，而原来的表格被删除。)

### 7.1.4 sql调优（携程）

1.选择正确的存储引擎

以 MySQL为例，包括有两个存储引擎 MyISAM 和 InnoDB，每个引擎都有利有弊。

MyISAM 适合于一些需要大量查询的应用，但其对于有大量写操作并不是很好。InnoDB 的趋势会是一个非常复杂的存储引擎，对于一些小的应用，它会比 MyISAM 还慢。但是它支持“行锁” ，于是在写操作比较多的时候，会更优秀。并且，他还支持更多的高级应用，比如：事务。

2.）优化字段的数据类型

记住一个原则，越小的列会越快。如果一个表只会有几列罢了（比如说字典表，配置表），那么，我们就没有理由使用 INT 来做主键，使用 MEDIUMINT, SMALLINT 或是更小的 TINYINT 会更经济一些。如果你不需要记录时间，使用 DATE 要比 DATETIME 好得多。当然，你也需要留够足够的扩展空间。

3.为搜索字段添加索引

索引并不一定就是给主键或是唯一的字段。如果在你的表中，有某个字段你总要会经常用来做搜索，那么最好是为其建立索引，除非你要搜索的字段是大的文本字段，那应该建立全文索引。

4.避免使用Select \*从数据库里读出越多的数据，那么查询就会变得越慢。并且，如果你的数据库服务器和WEB服务器是两台独立的服务器的话，这还会增加网络传输的负载。即使你要查询数据表的所有字段，也尽量不要用\*通配符，善用内置提供的字段排除定义也许能给带来更多的便利。

5.使用 ENUM 而不是 VARCHAR

ENUM 类型是非常快和紧凑的。在实际上，其保存的是 TINYINT，但其外表上显示为字符串。这样一来，用这个字段来做一些选项列表变得相当的完美。例如，性别、民族、部门和状态之类的这些字段的取值是有限而且固定的，那么，你应该使用 ENUM 而不是 VARCHAR。

6.尽可能的使用 NOT NULL

除非你有一个很特别的原因去使用 NULL 值，你应该总是让你的字段保持 NOT NULL。 NULL其实需要额外的空间，并且，在你进行比较的时候，你的程序会更复杂。 当然，这里并不是说你就不能使用NULL了，现实情况是很复杂的，依然会有些情况下，你需要使用NULL值。

7.固定长度的表会更快

固定长度的表会提高性能，因为MySQL搜寻得会更快一些，因为这些固定的长度是很容易计算下一个数据的偏移量的，所以读取的自然也会很快。而如果字段不是定长的，那么，每一次要找下一条的话，需要程序找到主键。

8.尽量避免以“%”开头的LIKE语句，模糊匹配，其会导致索引失效；

9.OR语句前后没有同时使用索引，会导致索引失效；

10.数据类型出现隐式转化（如varchar不加单引号的话可能会自动转换为int型），会导致索引失效；

11.应尽量避免在 where 子句中使用!=或<>操作符，否则将引擎放弃使用索引而进行全表扫描。

12.应尽量避免在 where 子句中对字段进行 null 值判断，否则将导致引擎放弃使用索引而进行全表扫描，如：

select id from t where num is null

13.应尽量避免在 where 子句中使用 or 来连接条件，否则将导致引擎放弃使用索引而进行全表扫描，如：

select id from t where num=10 or num=20

可以这样查询：

select id from t where num=10

union all

select id from t where num=20

14.in 和 not in 也要慎用，否则会导致全表扫描，如：

select id from t where num in(1,2,3)

对于连续的数值，能用 between 就不要用 in 了：

select id from t where num between 1 and 3

15.应尽量避免在 where 子句中对字段进行表达式操作，这将导致引擎放弃使用索引而进行全表扫描。如：

select id from t where num/2=100

应改为:

select id from t where num=100\*2

16.不要在 where 子句中的“=”左边进行函数、算术运算或其他表达式运算，否则系统将可能无法正确使用索引；

#### 7.1.4.1 count函数

count(\*)肯定是查全部的记录条数了

count(数字)，和count(\*)效果一样，有几条记录就显示几

count(属性名)则查处的是属性名下面的非null数据的个数

count()则报错 错误码: 1064

## 7.2 Redis(vipkid)

## 7.3 Memcached

## 7.4 框架

### 7.4.1 mybatis

#### 1.ResultMap和ResultType：

两者都是表示查询结果集与java对象之间的一种关系，处理查询结果集，映射到java对象。

resultMap：表示将查询结果集中的列一一映射到bean对象的各个属性。

ResultType:表示的是bean中的对象类，此时可以省略掉resultMap标签的映射，但是必须保证查询结果集中的属性 和 bean对象类中的属性是一一对应的，此时大小写不敏感，但是有限制。

#### 2.parameterType：

直接将查询结果列值类型自动对应到java对象属性类型上，不再配置映射关系一一对应。

#### 3.mybatis中的#和$的区别

#将传入的数据都当成一个字符串，会对自动传入的数据加一个双引号。

$将传入的数据直接显示生成在sql中。

#方式能够很大程度防止sql注入。$方式无法防止Sql注入。

一般能用#的就别用$.

MyBatis排序时使用order by 动态参数时需要注意，用$而不是#

#### 4.别名

<typeAliases>

<typeAlias alias="Author" type="domain.blog.Author"/>

</typeAliases>

或：  
@Alias("author")

public class Author {

...

}

#### 5.select

<select id="selectPerson" parameterType="int" resultType="hashmap">

SELECT \* FROM PERSON WHERE ID = #{id}

</select>

#### 6.insert

<insert

id="insertAuthor"

parameterType="domain.blog.Author"

useGeneratedKeys=""

timeout="20">

#### 7.单个关联查询，association标签

<resultMap id="blogResult" type="Blog">

<id property="id" column="blog\_id" />

<result property="title" column="blog\_title"/>

<association property="author" column="blog\_author\_id" javaType="Author" resultMap="authorResult"/>

</resultMap>

<resultMap id="authorResult" type="Author">

<id property="id" column="author\_id"/>

<result property="username" column="author\_username"/>

<result property="password" column="author\_password"/>

<result property="email" column="author\_email"/>

<result property="bio" column="author\_bio"/>

</resultMap>

#### 8.多个关联查询，collection标签 ：

<resultMap id="blogResult" type="Blog">

<id property="id" column="blog\_id" />

<result property="title" column="blog\_title"/>

<collection property="posts" ofType="Post">

<id property="id" column="post\_id"/>

<result property="subject" column="post\_subject"/>

<result property="body" column="post\_body"/>

</collection>

</resultMap>

#### 动态sql

1. If

<if test="title != null">

AND title like #{title}

</if>

1. choose,when,otherwise

<select id="findActiveBlogLike"

resultType="Blog">

SELECT \* FROM BLOG WHERE state = ‘ACTIVE’

<choose>

<when test="title != null">

AND title like #{title}

</when>

<when test="author != null and author.name != null">

AND author\_name like #{author.name}

</when>

<otherwise>

AND featured = 1

</otherwise>

</choose>

</select>

1. Where

<where>

<if test="state != null">

state = #{state}

</if>

<if test="title != null">

AND title like #{title}

</if>

<if test="author != null and author.name != null">

AND author\_name like #{author.name}

</if>

</where>

1. Set

<set>

<if test="username != null">username=#{username},</if>

</set>

1. Foreach

需要对一个集合进行遍历，通常是在构建 IN 条件语句的时候;

<select id="selectPostIn" resultType="domain.blog.Post">

SELECT \*

FROM POST P

WHERE ID in

<foreach item="item" index="index" collection="list"

open="(" separator="," close=")">

#{item}

</foreach>

</select>

## 7.5 NoSQL

## 7.6 Protobuf

## 7.7 Thrift

## 7.8Mongodb

## 7.9 数据库范式

### 7.9.1 第一范式

即数据库表的每一列都是不可分割的原子数据项，而不能是集合，数组，记录等非原子数据项。

### 7.9.2 第二范式

在1NF的基础上，非码属性必须完全依赖于候选码；

### 7.9.3 第三范式

在2NF基础上，任何非主属性不依赖于其它非主属性；

### 7.9.4 第四范式（巴斯-科德范式（BCNF））

在3NF基础上，任何非主属性不能对主键子集依赖

# 8.开发工具

## 8.1 IDEA：

### 8.1.1 快速生成常用方法：

1.alt+enter是一种用途广泛的操作，把光标放在一个没有get和set方法的私有成员变量上，按alt+enter就会弹出一个小窗；

2.光标在类里面，按alt+insert。弹出一个小窗，可以用它生成多种代码。

### 8.1.3 调试springboot代码：

1、打开Run/Debug Configurations,

2、选择Spring Boot 下要进行debug调试的服务,

3、在VM options:的填写框中写入"-Xms512m -Xmx512m -Xmn164m -XX:MaxPermSize=250m -XX:ReservedCodeCacheSize=64m -Dserver.port=8080 -ea",

4、port是该服务的端口号;Working directory选择$MODULE\_DIR$。

1. 然后debug运行微服务主程序入后即@springbootapplication注释的类就可以了

# Java多线程

## 创建线程

### 实现Runnable接口

实现接口中的run方法，其方法体中用来处理相关业务；

代码如下：

public class RunnableDemo implements Runnable*{*  
 private String name;  
 public RunnableDemo*(*String name*){* this.name = name;  
 System.*out*.println*(*"已创建线程"+name+"....."*)*;  
 *}* @Override  
 public void run*() {* System.*out*.println*(*"线程"+name+"开始运行....."*)*;  
 for*(*int i=4;i>0;i--*){* System.*out*.println*(*name+":"+i*)*;  
 try *{* Thread.*sleep(*50*)*;  
 *}* catch *(*InterruptedException e*) {* e.printStackTrace*()*;  
 *}  
 }* System.*out*.println*(*"线程"+name+"执行结束....."*)*;  
 *}*

### 继承Thread类

class threadServer extends Thread*{* @Override  
 public void run*(){  
 //业务代码   
 }  
}*

### 实现Callable接口，通过FutureTask包装器创建Thread线程

仅仅了解；

### 优缺点

优缺点：

1）、继承Thread类为单继承，实现Runnable方法为多实现，所以在灵活性上来说，使用实现Runnable方法更灵活；

2）、通过实现Runnable接口的方式可以实现多线程内的资源共享；

3）、增加代码的健壮性，代码可以被多个线程共享，代码和数据独立；

4）、线程池只能放实现Runnable或callable类的线程，不能直接放入继承Thread类的线程；

## 启动线程

通过Thread对象的start方法启动线程；

如果线程是通过继承Thread来创建的，则直接调用线程继承过来的start方法启动即可；

但是如果线程是通过实现Runnable接口来实现的，则需要使用Thread的有参构造方法来创建一个Thread对象，然后调用start方法启动线程，实现代码如下：

public void start*(){* System.*out*.println*(*"准备启动线程："+name*)*;  
 if*(*t == null*){* t = new Thread(this,name);  
 t.start*()*;  
 *}  
}*

## 常见方法：

### sleep方法：

sleep方法为Thread的静态方法；

sleep方法的作用是让线程休眠指定时间，在时间到达时自动恢复线程的执行；

sleep方法不会释放线程锁；

### wait方法

wait方法是Object的方法；

任意一个对象都可以调用wait方法，调用wait方法会将调用者的线程挂起，使该线程进入一个叫waitSet 的等待区域，直到其他线程调用同一个对象的notify方法方法才会重新激活调用者；

当wait方法被调用时，它会释放它所占用的锁标记，从而使线程所在对象中的synchronize数据可以被别的线程所使用，

所以wait（）方法必须在同步块中使用，notify（）和notifyAll（）方法都会对对象的“锁标记”进行修改，所以都需要在同步块中进行调用，

如果不在同步块中调用，虽然可以编辑通过，但是运行时会报IllegalMonitorStateException（非法的监控状态异常）；

### 9.3.3 yeild方法：

yeild（）方法表示停止当前线程，使该线程进入可执行状态，让同等优先级的线程运行，如果没有同等优先级的线程，那么yeild（）方法不会起作用；

### 9.3.4 notify和nofityAll方法

notify（）会通知一个处在wait（）状态的线程；如果有多个线程处在wait状态，他会随机唤醒其中一个；

notifyAll（）会通知过所有处在wait（）状态的线程，具体执行哪一个线程，根据优先级而定；

### 9.3.5 join（）方法

在当前线程中调用另一个线程的join（）方法，则当前线程转入阻塞状态，直到另一个线程运行结束，当前线程再由阻塞状态转为就绪状态；

## 停止线程

停止一个线程意味着在线程处理完任务之前停掉正在做的操作，即放弃当前的操作；

注意：如果停止正在睡眠的线程或者先停止线程，在运行线程睡眠的代码，则会抛出InterruptedException异常；

### for-break方式停止线程

代码实现如下：

@Override

public void run() {

super.run();

for (int i = 0; i < 100000; i++) {

if (this.isInterrupted()) {

System.out.println("已经是停止状态了！要退出了！");

break;

}

System.out.println("i=" + (i + 1));

}

System.out.println("我被输出，如果此代码是for又继续运行，线程并未停止！");

}

以上示例虽然停止了线程，但是如果for语句下面还有语句，那么还是会继续执行。

### 异常法停止线程

@Override

public void run() {

try {

for (int i = 0; i < 100000; i++) {

if (this.isInterrupted()) {

System.out.println("已经是停止状态了！要退出了！");

throw new Exception();

}

System.out.println("i=" + (i + 1));

}

System.out.println("我在for下面");

} catch (Exception e) {

System.out.println("进入类中run方法的catch了!");

e.printStackTrace();

}

}

用异常法来停止线程，就不会继续执行for后面的业务逻辑了；

### 使用return停止线程

@Override

public void run() {

while (true) {

if (this.isInterrupted()) {

System.out.println("停止了！");

return;

}

System.out.println("time=" + System.currentTimeMillis());

}

}

## Java线程状态

java线程有四种状态：产生、就绪、执行、阻塞、死亡

产生：线程被创建，但是未启动（未调用start（））

就绪：线程被启动（调用start（）），处于可执行状态，等待CPU的调度；

执行：线程正常的运行状态；

死亡：一个线程正常执行结束；

阻塞：

（1）、等待阻塞：运行的线程执行wait（）方法，jvm会把该线程放入waitSet线程池（释放锁）；

（2）、同步阻塞（死锁）：运行的线程在获取其他对象的同步锁是，该对象的同步锁被别的线程锁占用，则jvm会将该线程放入线程池中；

（3）、其他阻塞：运行的线程执行sleep方法或者执行t.join（）方法，被别的线程打断，jvm把该线程置为阻塞状态，当sleep超时或者join线程结束时线程重新进入就绪状态；

## 线程同步

线程同步主要使用synchronized关键字；具体有两种实现方式：

1. 作为关键字修饰类中一个方法；
2. 修饰方法中的一块区域(代码);

## 线程之间的数据传递

### 9.7.1

### 9.7.2 通过构造方法传递数据：

package mythread;

public class MyThread1 extends Thread

{

private String name;

public MyThread1(String name)

{

this.name = name;

}

public void run()

{

System.out.println("hello " + name);

}

}

这种方法是在创建线程对象的同时传递数据的，因此，在线程运行之前这些数据就就已经到位了，这样就不会造成数据在线程运行后才传入的现象。

如果要传递更复杂的数据，可以使用集合、类等数据结构。

使用构造方法来传递数据虽然比较安全，但如果要传递的数据比较多时，就会造成很多不便。

### 9.7.3 通过变量和方法传递数据

package mythread;

public class MyThread2 implements Runnable

{

private String name;

public void setName(String name)

{

this.name = name;

}

public void run()

{

System.out.println("hello " + name);

}

}

### 9.7.4 线程之间的数据共享，以下贴出生产者消费者模式的实现：

实现场景：有一个馒头房，生产者生产馒头，消费者消费馒头，当馒头数量为0时要停止消费，开始生产，当生产的馒头大于5时停止生产，开始消费；

实现代码如下：

# 网站优化

## 网站访问速度优化:10.(vipkid)

## 10.2网络安全：

### 10.2.1 同源策略

源：协议、域名、端口

同源：若地址中的协议、域名、端口都相同，即为同源。反之，有一者不同即为不同源。

同源策略：为浏览器的一个安全功能，不同源的客户端脚本没有明确授权的情况下，无法读写对方的资源。

### 10.2.2 Ajax是遵循同源策略；

### 10.2.3 CSP是什么？如何设置CSP？

CSP：Content Security Policy，内容安全策略。

其思想是以服务器白名单的形式来配置可信的内容来源，客户端Web应用代码可以使用这些安全来源。

### 10.2.4 防御CSRF都有哪些方法，JAVA是如何防御CSRF漏洞的，token一定有用么？

CSRF是跨站请求伪造，是通过劫持其他用户身份信息，进行恶意操作。

防御方法：

验证HttpRefer字段

在请求地址中添加token并验证

在Http头中自定义属性并验证

token不一定有用；

基于token的身份验证流程如下：

客户端使用用户名跟密码请求登录

服务端收到请求，去验证用户名与密码

验证成功后，服务端会签发一个 Token，再把这个 Token 发送给客户端

客户端收到 Token 以后可以把它存储起来，比如放在 Cookie；

客户端每次向服务端请求资源的时候需要带着服务端签发的 Token；

服务端收到请求，然后去验证客户端请求里面带着的 Token，如果验证成功，就向客户端返回请求的数据；

### 10.2.5 XSS攻击与防御

跨站脚本攻击简称 XSS (Cross-Site Scriptting)，是一种经常出现在web应用中的计算机安全漏洞，它允许恶意web用户将代码植入到提供给其它用户使用的页面中。

XSS攻击的危害：

1、盗取各类用户帐号权限(控制所盗窃权限数据内容)，如机器登录帐号、用户网银帐号、各类管理员帐号

2、控制企业数据，包括读取、篡改、添加、删除企业敏感数据的能力

3、基于XSS的跨站业务请求(如：非法转账、非法下单、非法转载/发表内容、发送电子邮件、利用管理员身份提权挂马、控制受害者机器向其它网站发起攻击等)

防御措施：

- 避免使用客户端数据，尽量服务端生成

- JavaScript变量输出时先编码

- XSS过滤（js-xss, http://jsxss.com）

- CSP(Content Security Policy)

## 10.3高并发性能解决方案（贝壳找房）

## 10.4 数据安全解决方案（贝壳找房）

# 11.分布式存储计算框架（小米）

# 12.web服务器

## 12.1 tomcat

## 12.2 Nginx（网易游戏）

# 13.前端基础

## 13.1 /HTML/JavaScript/CSS/HTML5（咪咕）

## 13.2 W3C标准

# 14.网络编程

## 14.1socket网络编程

## 14.2 路由汇聚

路由汇聚就是找到两路由不同地方，换算成二进制后找到能匹配的最大长度，则能匹配的最大长度再换算成的十进制便是汇聚后的结果，新掩码的最大长度= 原掩码长度-不匹配的长度。

# 脚本

## 15.1 shell

## 15.2 python

# 16.系统

## 16.1 64位：

### 16.1.1 虚拟进程地址空间

64位操作系统会为每个进程分配一个 8GB 大小的虚拟进程地址空间；

## 16.2 32位：

### 16.2.1 虚拟进程地址空间

32位操作系统会为每个进程分配一个 4GB 大小的虚拟进程地址空间；

## 16.3 Linux：

### 16.3.1 Linux进程间通信方式：

1.管道（pipe）,流管道(s\_pipe)和有名管道（FIFO）

管道这样的通讯方式有两种限制，一是半双工的通信，数据仅仅能单向流动，二是仅仅能在具有亲缘关系的进程间使用。进程的亲缘关系一般是指父子进程关系。

流管道s\_pipe: 去除了第一种限制,能够双向传输.

命名管道:name\_pipe克服了管道没有名字的限制，因此，除具有管道所具有的功能外。它还同意无亲缘关系进程间的通信；

2.套接字

3.信号量

4.文件锁

5.共享内存

6.消息队列

## 16.4 死锁

### 16.4.1 简介：

死锁是由于两个或以上的线程互相持有对方需要的资源，导致这些线程处于等待状态，无法执行。

### 16.4.2 产生死锁的四个必要条件

1.互斥性：线程对资源的占有是排他性的，一个资源只能被一个线程占有，直到释放。

2.请求和保持条件：一个线程对请求被占有资源发生阻塞时，对已经获得的资源不释放。

3.不剥夺：一个线程在释放资源之前，其他的线程无法剥夺占用。

4.循环等待：发生死锁时，线程进入死循环，永久阻塞。

### 16.4.3 避免死锁的方法

1.破坏“请求和保持”条件

　　　　想办法，让进程不要那么贪心，自己已经有了资源就不要去竞争那些不可抢占的资源。比如，让进程在申请资源时，一次性申请所有需要用到的资源，不要一次一次来申请，当申请的资源有一些没空，那就让线程等待。不过这个方法比较浪费资源，进程可能经常处于饥饿状态。还有一种方法是，要求进程在申请资源前，要释放自己拥有的资源。

2.破坏“不可抢占”条件

　　　　允许进程进行抢占，方法一：如果去抢资源，被拒绝，就释放自己的资源。方法二：操作系统允许抢，只要你优先级大，可以抢到。

3.破坏“循环等待”条件

　　　　将系统中的所有资源统一编号，进程可在任何时刻提出资源申请，但所有申请必须按照资源的编号顺序（升序）提出；

### 16.4.4 死锁相关算法

1.银行家算法（避免死锁）：

银行家算法是从当前状态出发，按照系统各类资源剩余量逐个检查各进程需要申请的资源量，找到一个各类资源申请量均小于等于系统剩余资源量的进程P1。然后分配给该P1进程所请求的资源，假定P1完成工作后归还其占有的所有资源，更新系统剩余资源状态并且移除进程列表中的P1，进而检查下一个能完成工作的客户，......。如果所有客户都能完成工作，则找到一个安全序列，银行家才是安全的。若找不到这样的安全序列，则当前状态不安全。

2.资源有序分配法：预防死锁

3.资源分配图化简法：检测死锁

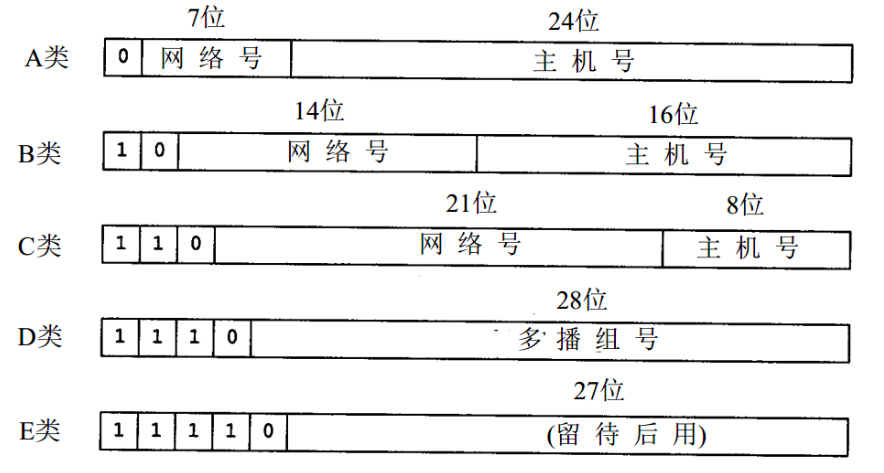
4.撤销进程法：解决死锁

## 16.5 文件管理：

扇区是磁盘最小的物理存储单元，一般而言是每个扇区512B大小，但是操作通常不直接管理每一个扇区，而是通过将若干个扇区组成的一个更大的集合来去进行操作管理。

这个比扇区更大的集合，在Windows下叫做簇；在Linux下叫做块（block）。

## 16.6 ip地址分类



## 16.7 创建进程的步骤：

1.申请空白的PCB(进程控制块)；

2.为新进程分配资源；

3.初始化PCB；

4.将新进程插入就绪队列；

## 16.8 进程和线程的区别：

1，进程：子进程是父进程的复制品。子进程获得父进程数据空间、堆和栈的复制品。

2，线程：相对与进程而言，线程是一个更加接近与执行体的概念，它可以与同进程的其他线程共享数据，但拥有自己的栈空间，拥有独立的执行序列。

两者都可以提高程序的并发度，提高程序运行效率和响应时间。

线程和进程在使用上各有优缺点：线程执行开销小，但不利于资源管理和保护；而进程正相反。同时，线程适合于在SMP机器上运行，而进程则可以跨机器迁移。

根本区别就一点：用多进程每个进程有自己的地址空间(address space)，线程则共享地址空间。所有其它区别都是由此而来的：

1、速度：线程产生的速度快，线程间的通讯快、切换快等，因为他们在同一个地址空间内。

2、资源利用率：线程的资源利用率比较好也是因为他们在同一个地址空间内。

3、同步问题：线程使用公共变量/内存时需要使用同步机制还是因为他们在同一个地址空间内

# 17 C/C++语言：

## c++类占用字节：

空类求sizeof为1，加不加构造函数对sizeof没影响，但有了虚函数，则需要有一个指针指向虚函数表，32位下，指针sizeof为4；

注意：此时如果类中有一个char，该类占用字节数也是4，因为此时虚函数指针所申请到的字节数足以存储char；

## 大端/小端机器存储行为：

结构体中，先声明的是低地址，后声明的是高地址

大端：低位放在高地址：

小端：高位放在高地址：

## 17.3析构函数

在C++中，析构函数的作用是：当一个对象被销毁时，调用析构函数对类对象和对象成员进行释放内存资源。

当我们定义一个指向派生类类型对象指针时，构造函数按照从基类到派生类的顺序被调用，但是当删除指向派生类的基类指针时，派生类的析构函数没有被调用，只是调用了基类的析构函数，此时派生类将会导致内存泄漏；（解决方法：将基类的析构函数声明为虚析构函数）

## 指针：

### 17.3.1 指针数组和数组指针

int \*p[4];  //定义一个指针数组，该数组中每个元素是一个指针，每个指针指向哪里就需要程序中后续再定义了。

int (\*p)[4];  //定义一个数组指针，该指针指向含4个元素的一维数组（数组中每个元素是int型）。

### 17.3.2 指针占有的字节数

指针跟自己的系统有关，32位系统是4个字节，64是8个字节；

## 内存对齐原则

第一个是 4,1,2 内存对齐为4+2+2=8；

第二个是1,4,2 内存对齐为 4+4+4=12

## 枚举

枚举赋值问题，从上一个定义开始逐个低增；

## 虚函数

C++中 的虚函数的作用主要是实现了多态的机制。而虚函数是通过虚函数表(V-Table)实现的。

构造函数不能声明为虚函数，析构函数可以声明为虚函数，而且有时是必须声明为虚函数。

构造函数为什么不能声明为虚函数？

1 构造一个对象的时候，必须知道对象的实际类型，而虚函数行为是在运行期间确定实际类型的。而在构造一个对象时，由于对象还未构造成功。编译器无法知道对象的实际类型，是该类本身，还是该类的一个派生类，或是更深层次的派生类。无法确定。

2 虚函数的执行依赖于虚函数表。而虚函数表在构造函数中进行初始化工作，即初始化vptr，让他指向正确的虚函数表。而在构造对象期间，虚函数表还没有被初 始化，将无法进行。 析构函数执行时先调用派生类的析构函数，其次才调用基类的析构函数。

3.析构函数为什么声明为虚函数？

如果析构函数不是虚函数，而程序执行时又要通过基类的指针去销毁派生类的动态对象，那么用delete销毁对象时，只调用了基类的析构函数，未调用派生类的析构函数。这样会造成销毁对象不完全。

包含至少一个纯虚函数的类视为抽象类

## 有可能造成缓冲区溢出的函数：

1、strcpy()：字符串复制

2、strcat()：字符串连接

3、gets()：从输入流读取字符串到缓冲区

4、sprintf()：打印字符串到缓冲区

5、vsprintf()：打印字符串到缓冲区

6、scanf系列：从输入流读取格式化字符

7、strdup()：字符串复制

# 18 常见协议

## 18.1 TCP协议：

### 18.1.1 功能

有连接、数据可靠传输、端到端的流量控制；

### 18.1.2 所支持的协议

Telnet：端口号是23，远程登录控制Web服务器；

FTP：端口号是21和20，文件传输协议；

SMTP：端口号是25，邮件传送协议；

## 18.2 UDP协议

### 18.2.1 简介：

用户数据报协议，无连接、不可靠、快速传输，发送IP数据包；

### 18.2.2 所支持的协议：

NFS（网络文件系统）：端口号是2049，通过使用NFS，用户和程序可以像访问本地文件一样访问远端系统上的文件；

SNMP（简单网络管理协议）：端口号默认161或162；

DNS（域名解析协议）：端口号是53；

TFTP（通用文件传输协议）：端口号是69；

## 18.3 ping命令没有涉及到的网络协议是：

ping后面跟的是地址，所以要先将域名转换为ip地址，即用到了DNS 获取到ip地址后，在数据链路层是根据MAC地址传输的，所以要用到ARP解析服务，获取到MAC地址 ping功能是测试另一台主机是否可达，程序发送一份ICMP回显请求给目标主机，并等待返回ICMP回显应答，（ICMP主要是用于ip主机、路由器之间传递控制信息，控制信息是指网络通不通，主机是否科大） TCP的话，不涉及数据传输，不会用到；

# 19.jvm

## 1.内存划分：

### 1.堆：

用于存储各类对象；堆区被细化可以分为年轻代、老年代，而年轻代又可分为Eden区、From Survivor、To Survivor三个区域，

## 2.方法区

方法区主要是存储类的元数据的，如虚拟机加载的类信息、编译后的代码等。

### 3.虚拟机栈

Java虚拟机栈（Java Virtual Machine Stacks）也是线程私有的，它的生命周期与线程相同。每个方法被执行的时候都会同时创建一个栈帧（Stack Frame）用于存储局部变量表、操作栈、动态链接、方法出口等信息。每一个方法被调用直至执行完成的过程，就对应着一个栈帧在虚拟机栈中从入栈到出栈的过程。

### 4.本地方法栈

这部分区域是专门为Native方法来实现的

### 5.程序计数器

字节码解释器工作时就是通过改变这个计数器的值来选取下一条需要执行的字节码指令，分支、循环、跳转、异常处理、线程恢复等基础功能都需要依赖这个计数器来完成。

由于Java虚拟机的多线程是通过线程轮流切换并分配处理器执行时间的方式来实现的，在任何一个确定的时刻，一个处理器（对于多核处理器来说是一个内核）只会执行一条线程中的指令。因此，为了线程切换后能恢复到正确的执行位置，每条线程都需要有一个独立的程序计数器，各条线程之间的计数器互不影响，独立存储，我们称这类内存区域为“线程私有”的内存。

如果线程正在执行的是一个Java方法，这个计数器记录的是正在执行的虚拟机字节码指令的地址；如果正在执行的是Natvie方法，这个计数器值则为空（Undefined）。

\*\*\*光峰科技

